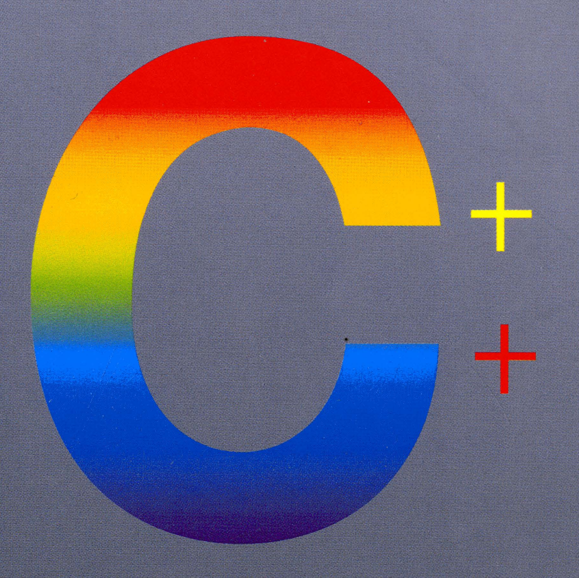
LE LANGAGE C++

Cours détaillé et exercices corrigés



Baghdadi ZITOUNI

 Centre de Publication Universitaire

Le langage C++

Cours détaillé & exercices corrigés

Baghdadi Zitouni

Technologue à l'ISET de Ksar-Hellal

Centre de Publication Universitaire

2011

B.P: 223 La Manouba 2010 - TUNISIE
Tél: (216) 71 600 025 Fax: (216) 71 601 266

Avant propos

Cet ouvrage intitulé « Le langage C++ » est le résultat d'environ 15 années d'enseignements dans plusieurs institutions universitaires. De ce fait, il a bénéficié des questions et problèmes posés par les étudiants durant les cours et travaux pratiques de programmation.

L'objectif de l'ouvrage est d'expliquer en termes simples et à travers de nombreux exemples et exercices corrigés comment utiliser l'approche orientée objet pour développer des programmes en C++ afin de résoudre différents types de problèmes et manipuler diverses structures de données statiques et dynamiques.

La première partie du livre présente les fondements du langage C++ : la structure d'un programme, les types prédéfinis, les variables, les opérateurs, etc.

La deuxième partie présente en détail les structures de contrôle conditionnelles et itératives ainsi que les différents types de fonctions. Des applications sont faites sur des tableaux, des chaînes de caractères, des pointeurs, etc.

La troisième partie est consacrée à la programmation orientée objet. On y présente les classes, les objets, la surcharge des opérateurs, l'héritage, le polymorphisme, etc.

La quatrième partie traite les structures de données dynamiques (listes chainées, piles, files) et les fichiers en utilisant toujours l'approche orientée objet.

Le livre contient environ une centaine d'exercices souvent conçus comme une application du cours à des situations de la vie professionnelle (calcul mathématique, problèmes de gestion, jeux, etc.).

La solution proposée à chaque exercice n'est pas unique et dans plusieurs cas elle n'est pas optimale car on a toujours privilégié l'apport pédagogique et la simplicité.

Tous les programmes qui figurent dans l'ouvrage ont été testés à l'aide du compilateur Visual C++ sous Windows XP. De ce fait, ils peuvent nécessiter certaines rectifications en cas de changement de plateforme.

Je tiens à remercier toutes les personnes qui m'ont aidé à l'écriture de cet ouvrage et, en particulier, mes collègues Kamel Ben Rhouma, Chiheb Chaieb, Kais Debbabi et Lobna Ben Tekaya pour leur active et aimable participation.

Pour signaler une erreur ou faire des remarques et/ou suggestions, veuillez envoyer un e-mail à l'adresse :

baghdadi.zitouni@gmail.com

Introduction

Petite histoire du C

Le langage C a été mis au point par Dennis Ritchie et Brian Kernighan au début des années 70. Le but était de développer un langage qui permettrait d'obtenir un système d'exploitation de type UNIX portable. La première édition de la définition de ce langage a été donnée dans leur livre commun "The C programming language".

Suite à l'apparition de nombreux compilateurs C, l'ANSI (American National Standards Institute) a décidé de normaliser ce langage pour donner ce que l'on appelle le C-ANSI.

Le langage C est actuellement l'un des langages les plus utilisés étant donné les multiples qualités qu'il possède :

- c'est un langage **très rapide** car assez bas niveau ;
- c'est un langage **portable**. Le même code source peut être compilé aussi bien sous Windows, Linux, Mac OS et ne dépend d'aucun type de processeur particulier ;
- il met en œuvre un nombre restreint de concepts, ce qui facilite sa maîtrise et l'écriture de compilateurs simples et rapides.

Toutefois, le langage C possède plusieurs défauts :

- il offre peu de vérifications lors de la compilation et n'offre aucune vérification pendant l'exécution, ce qui fait que des erreurs qui pourraient être automatiquement détectées lors du développement ne le sont que plus tard, souvent au prix d'un plantage du logiciel ;
- il n'offre pas de support direct à des concepts informatiques plus modernes comme la programmation orientée objet ou la gestion d'exceptions ;
- le support de l'allocation de mémoire et des chaînes de caractères est minimaliste, ce qui oblige les programmeurs à s'occuper de détails fastidieux et sources de bugs.

Histoire du C++

Le langage C++ est une « amélioration » du langage C. Bjarne Stroustrup, un ingénieur considéré comme l'inventeur du C++, a décidé d'ajouter au langage C les propriétés de l'*approche orientée objet.* Ainsi, vers la fin des années 80 un nouveau langage, baptisé *C with classes*, fut apparaitre. Celui-ci a ensuite été renommé en C++, clin d'œil au symbole d'incrémentation ++ du langage C, afin de signaler qu'il s'agit d'un langage C amélioré.

Le C++ est actuellement l'un des langages de programmation les plus utilisés dans le monde. Il est à la fois facile à utiliser et très efficace. Ce langage permet la programmation sous de multiples paradigmes comme la programmation procédurale, la programmation orientée objet et la programmation générique. Ce langage n'appartient à personne et par conséquent n'importe qui peut l'utiliser sans avoir besoin d'une autorisation ou obligation de payer pour avoir le droit d'utilisation.

Les améliorations du C++

Le C++ reprend la quasi-intégralité des concepts présents dans le langage C, si bien que les programmes écrits en langage C fonctionnent avec un compilateur C++.

Les améliorations apportées par Bjarne Stroustrup concernent la prise en charge des classes et des autres concepts objet tels que l'encapsulation, l'héritage (simple et multiple) et le polymorphisme.

Parmi les autres fonctionnalités ajoutées au C++, on peut citer :

- les arguments par défaut ;
- la surcharge des fonctions et des opérateurs ;
- les fonctions virtuelles et les fonctions inline ;
- la programmation générique grâce aux « *templates* » ;
- la gestion des exceptions ;
- les espaces de noms (*namespaces*).

Notons enfin que le langage C++ est normalisé par l'ISO. Sa première normalisation date de 1998, sa dernière date de 2003 (ISO/CEI 14882:2003).

Fondements du langage C++

1. Mise en œuvre d'un programme C++

La réalisation d'un programme commence par la traduction en C++ des algorithmes. Cette phase de rédaction se fait à l'aide d'un **éditeur de texte**. Le programme source ainsi rédigé est ensuite traité par le **préprocesseur** qui, suivant les directives de compilation, substitue des symboles, inclut d'autres fichiers sources, etc. Le fichier source d'origine est retravaillé puis présenté au **compilateur**. Le compilateur examine le programme ligne par ligne, vérifie la syntaxe et génère un code intermédiaire (code objet). Il reste alors à rassembler les codes intermédiaires (dans le cas où le programme est découpé en plusieurs fichiers sources) et à les lier aux diverses librairies. C'est le rôle de **l'éditeur de liens** (linker) qui produit le code exécutable du programme (voir figure 1).

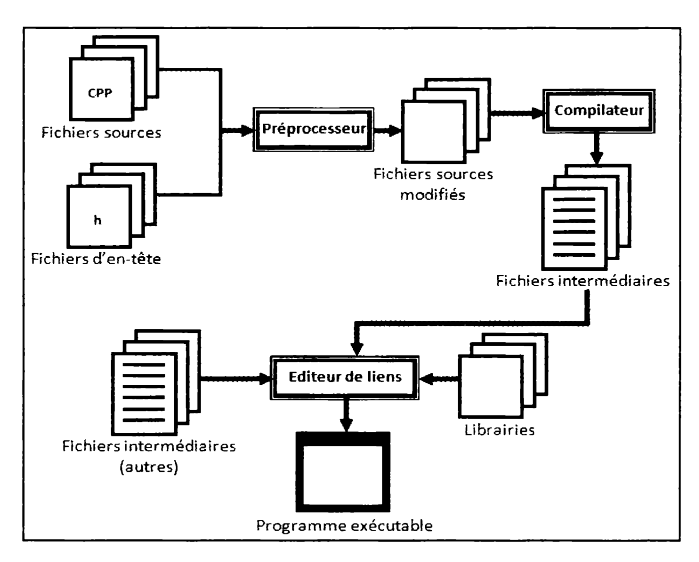


Figure 1 : Mise en œuvre d'un programme en C++

Il existe plusieurs environnements fournissant un éditeur de texte, un compilateur et un éditeur de liens. Ces environnements sont appelés **EDI** (Environnement de Développement Intégré).

Les principaux EDI permettant le développement d'applications en langage C++ sont :

- Borland C++
- Microsoft Visual C++
- NetBeans.

2. Structure d'un programme en C++

Probablement la meilleure manière de commencer à apprendre un langage de programmation est en écrivant un programme. Par conséquent, voici notre premier programme en C++ :

```
// Premier programme en C++
#include <iostream>
using namespace std;

int main( )
{
      cout<<"Bonjour tout le monde !"<<endl;
      return 0;
}
```

Output du programme

```
Bonjour tout le monde !
```

Le programme précédent est un programme typique que les programmeurs débutants écrivent pour la première fois. C'est l'un des programmes les plus simples qui peuvent être écrits en C++, mais il contient toutefois les éléments fondamentaux que chaque programme C++ en a.

Nous allons examiner ligne par ligne le code de ce programme :

```
// Premier programme en C++
```

C'est une ligne de commentaire. Toutes les lignes commençant par deux slashs (//) sont considérées comme des commentaires et n'ont aucun effet sur le comportement du programme. Lorsqu'un programme devient long et compliqué, il est conseillé d'y insérer des commentaires afin d'en expliquer le fonctionnement.

Un commentaire peut s'étendre sur plusieurs lignes. Dans ce cas, il doit être mis entre les délimiteurs /* et */ comme dans l'exemple suivant :

```
/* Test.cpp
Auteur: B. Zitouni
Date: 28-10-2010 */
```

```
#include <iostream>
```

Les lignes commençant par le symbole # sont des directives destinées au préprocesseur. Dans notre cas, cette ligne demande au préprocesseur d'inclure le fichier d'entrée-sortie standard *iostream*.

```
using namespace std;
```

Tous les éléments de la bibliothèque standard de C++ sont rassemblés dans un espace de nom (namespace) appelé *std*. Cette déclaration nous permet d'accéder et tirer profit de ces éléments.

Cette ligne est très fréquente dans les programmes C++ qui emploient la bibliothèque standard. Elle sera incluse dans tous les programmes qui figurent dans cet ouvrage.

```
int main()
```

Cette ligne marque le début de la définition de la fonction principale qui constitue le point d'entrée à partir duquel commence l'exécution des programmes C++. D'autres fonctions peuvent être définies avant ou après la fonction main( ), mais dans tous les cas, les instructions de cette fonction seront toujours les premières à être exécutées.

Le nom de la fonction doit être suivi d'une paire de parenthèses qui contient la liste des paramètres ou vide comme dans le cas présent.

Le mot *int* signifie que la fonction est censée retourner un résultat de type entier.

En C++, le corps de n'importe quelle fonction est toujours mis entre deux accolades { et }.

```
cout<<"Bonjour tout le monde !"<<endl;
```

Cette ligne est une instruction de C++. Une instruction est une expression simple ou composée qui peut réellement produire un certain effet.

Le *cout* représente le « flot de sortie » standard dans C++ qui est souvent associé à l'écran, d'où l'effet de cette instruction qui va afficher le message « Bonjour tout le monde ! ».

Le manipulateur *endl* permet de signaler une fin de ligne et d'effectuer un saut de ligne lorsqu'il est employé sur un flux de sortie.

Le *cout* est déclaré dans le fichier standard *iostream* du namespace *std*, c'est pourquoi nous avons inclus ce fichier et nous avons déclaré que nous allons utiliser l'espace de nom *std* au début de notre programme.

```
return 0;
```

Cette instruction fait renvoyer le code de retour 0 au système d'exploitation pour lui indiquer que le programme s'est exécuté sans aucune erreur.

Remarques

1. Le langage C++ est sensible à la casse, cela signifie qu'un nom contenant des majuscules est différent du même nom écrit en minuscules. Ainsi, les spécifications du langage exigent que la fonction principale soit appelée *main()* et non pas *Main()* ou *MAIN()*.

2. En C++, toute instruction se termine par le caractère point-virgule « ; ».

3. La division du programme en plusieurs blocs et l'indentation des instructions de chaque bloc offrent une meilleure lisibilité au programme et le rendent, par conséquent, plus facile à lire et à déboguer.

3. Entrées/Sorties simples

Un programme qui utilise les flux standards d'entrée-sortie doit comporter les directives suivantes :

```
#include <iostream>
using namespace std;
```

Les flux d'entrée-sortie sont représentés dans les programmes par les trois variables pré-déclarées et pré-initialisées suivantes :

- **cin**, le flux standard d'entrée, qui est habituellement associé au clavier du poste de travail ;

- **cout**, le flux standard de sortie, qui est habituellement associé à l'écran du poste de travail ;

- **cerr**, le flux standard pour la sortie des messages d'erreur, également associé à l'écran du poste de travail.

Les écritures et lectures sur ces unités ne se font pas en appelant des fonctions, mais à l'aide des opérateurs <<, appelé *opérateur d'injection* (« injection » de données dans un flux de sortie), et >>, appelé *opérateur d'extraction* (« extraction » de données d'un flux d'entrée). Grâce au mécanisme de surcharge des opérateurs qui permet la détection des types des données à lire ou à écrire, le programmeur n'a plus à se soucier des spécifications des formats de lecture et d'écriture.

La syntaxe d'une injection de données sur la sortie standard *cout* est :

```
cout<<expression1<<expression2<<... ;
```

La syntaxe d'une extraction de données à partir de l'entrée standard *cin* est :

```
cin>>variable1>>variable2>>... ;
```

A titre d'exemple, le programme suivant permet de lire les valeurs de deux entiers a et b à partir du clavier puis il affiche leur somme à l'écran.

```
#include <iostream>
using namespace std;

int main( )
{
      int a, b;
      cout<<"Entrer la valeur de a : ";
      cin>>a;
      cout<<"Entrer la valeur de b : ";
      cin>>b;
      cout<<a<<" + "<<b<<" = "<<a+b<<endl;
      return 0;
}
```

Test du programme

```
Entrer la valeur de a : 9
Entrer la valeur de b : 8
9 + 8 = 17
```

Les types de données

1. La notion de type

Les données manipulées en langage C++, comme en langage C, sont typées, c'est-à-dire qu'il faut préciser leur type dès le début, ce qui permet de connaître l'occupation mémoire (le nombre d'octets) de la donnée ainsi que sa représentation.

Le tableau suivant fournit une description des principaux types de données disponibles en C++ :

Tableau 1 : Types de données prédéfinis en C++

Type de données	Signification	Taille* (en octets)	Plage de valeurs acceptée
char	caractère	1	-128 à 127
unsigned char	caractère non signé	1	0 à 255
int	entier	4	-2 147 483 648 à 2 147 483 647
short int	entier court	2	-32 768 à 32 767
unsigned short int	entier court non signé	2	0 à 65 535
unsigned int	entier non signé	4	0 à 4 294 967 295
long int	entier long	4	-2 147 483 648 à 2 147 483 647
unsigned long int	entier long non signé	4	0 à 4 294 967 295
float	flottant (réel)	4	$-3.4*10^{38}$ à $3.4*10^{38}$
double	flottant double	8	$-1.7*10^{308}$ à $1.7*10^{308}$
long double	flottant double long	10	$-3.4*10^{4932}$ à $3.4*10^{4932}$
bool	booléen	1	true (1) / false (0). Toute valeur autre que 0 est considérée comme égale à true.
wchar_t	caractère étendu	2	caractère étendu (Unicode)

Remarque*

En réalité, l'espace mémoire alloué à une variable varie selon le système d'exploitation et le compilateur utilisés. Les valeurs mentionnées dans le tableau correspondent à la plupart des systèmes 32-bits. Pour connaitre la taille réelle allouée à un type particulier, on peut utiliser la fonction ***sizeof( )*** comme dans l'exemple suivant :

```
cout<<sizeof(int);
```

2. Le type int (nombre entier)

Un nombre entier est un nombre sans virgule (sans partie décimale) qui peut être exprimé dans différentes bases :

- *Base décimale :* l'entier est représenté par une suite de chiffres unitaires (de 0 à 9) ne devant pas commencer par le chiffre 0 ;
- *Base hexadécimale* : l'entier est représenté par une suite d'unités (de 0 à 9 ou de A à F (ou a à f)) devant commencer par 0x ou 0X ;
- *Base octale* : l'entier est représenté par une suite d'unités (incluant uniquement des chiffres de 0 à 7) devant commencer par 0.

Exemple

```
#include <iostream>
using namespace std;

int main( )
{
	int a = 10, b = 0XF, c = 010;
	cout<<"a = "<<a<<endl;
	cout<<"b = "<<b<<endl;
	cout<<"c = "<<c<<endl;
	return 0;
}
```

Output du programme

```
a = 10
b = 15
c = 8
```

3. Le type float (nombre réel)

Un nombre à virgule flottante (réel) peut être représenté de différentes façons :

- un entier décimal : 895
- un nombre comportant un point (et non une virgule) : 845.32
- une fraction : 27/11
- un nombre exprimé en puissance de 10 qui se note de la façon suivante :

mantisse e | E exposant

Exemples

- le nombre 12.3e4 correspond à 12.3×10^4 (soit 123000)
- le nombre 12.3E-3 correspond à 12.3×10^{-3} (soit 0.0123)

4. Le type char (caractère)

Le type *char* permet de stocker la valeur ASCII d'un caractère, c'est-à-dire un nombre entier (voir annexe). A titre d'exemple, si on désire stocker la lettre B (son code ASCII est 66) dans une variable, on pourra définir cette donnée soit par le nombre 66, soit en notant 'B' où les apostrophes simples signifient *code ascii de.*

Il n'existe pas de type de données standard pour les chaînes de caractères en C++. Pour créer une chaîne de caractères, on utilisera un tableau contenant dans chacune de ses cases un caractère ou on fait appel aux services de la classe *string* (voir page 127).

5. Création d'un type de données

Il est possible en C++ de définir un nouveau type de données grâce au mot clé ***typedef***. Celui-ci admet la syntaxe suivante :

typedef Caractéristiques_du_type Nom_du_type

où

- *Caracteristiques_du_type* représente un type de données existant (par exemple *float*, *short int*, ...) ;
- *Nom_du_type* définit le nom à attribuer au nouveau type de données.

Ainsi, l'instruction suivante crée un type de données *mot* calqué sur le type unsigned int :

```
typedef unsigned int mot;
```

6. Conversion de type de données

On appelle *conversion de type de données* le fait de modifier le type d'une donnée en un autre. Imaginons que l'on travaille par exemple sur une variable en virgule flottante (type *float*) et que l'on veuille « supprimer les chiffres après la virgule », c'est-à-dire convertir un *float* en *int*. Une telle opération peut être réalisée de deux manières :

- **conversion implicite** : si une valeur d'un type donné est affectée à une variable de type différent, le compilateur ne signalera aucune erreur mais effectuera une conversion automatique de la donnée avant de l'affecter à la variable.

Exemple

```
int x;
x = 8.324;
```

Après affectation, x contiendra la valeur 8.

- **conversion explicite** : une conversion explicite (appelée aussi *opération de cast*) consiste en une modification forcée du type de données. On utilise à cet effet un opérateur dit *de cast* pour spécifier la conversion. L'opérateur de cast est tout simplement le type de données, dans lequel on désire convertir la donnée, mis entre parenthèses avant la valeur ou le nom de la variable à convertir.

Exemple

```
int x;
x = (int)8.324;
```

Après affectation, x contiendra la valeur 8.

De plus, le langage C++ rajoute une notation fonctionnelle pour faire une conversion explicite.

Exemple

```
int x;
x = int(66.384);
```

Exercice

Qu'affichera le programme suivant :

```
#include <iostream>
using namespace std;

int main( )
{
      int n = 66;
      char c = 'B';
      cout<<"n = "<<(char)n<<endl;
      cout<<"c = "<<(int)c<<endl;
      return 0;
}
```

Output du programme

```
n = B
c = 66
```

Les variables

1. La notion de variable

Une variable est un objet repéré par son nom, pouvant contenir des données, qui pourront être modifiées lors de l'exécution du programme. Les variables en langage C++ sont ***typées***, c'est-à-dire que les données contenues dans celles-ci possèdent un type. Ainsi, elles sont stockées à une adresse mémoire et occupent un nombre d'octets dépendant de leur type.

En langage C++, les noms de variables peuvent être aussi long que l'on désire, toutefois le compilateur ne tiendra compte que des 32 premiers caractères. De plus, ces noms doivent répondre aux trois critères suivants :

1. un nom de variable peut comporter des lettres (majuscules ou minuscules), des chiffres et des traits de soulignement « _ » (les espaces et les autres caractères spéciaux ne sont pas autorisés) ;
2. un nom de variable doit commencer par une lettre ou un trait de soulignement (pas par un chiffre) ;
3. les noms de variables doivent être différents des mots qui figurent sur le tableau 2 (qui sont des mots-clés réservés) :

Tableau 2 : Mots réservés du langage C++

asm	auto	bool	break	case
catch	char	class	const	const_cast
continue	default	delete	do	double
dynamic_cast	else	enum	explicit	export
extern	false	float	for	friend
goto	if	inline	int	long
mutable	namespace	new	operator	private
protected	public	register	reinterpret_cast	return
short	signed	sizeof	static	static_cast
struct	switch	template	this	throw
true	try	typedef	typeid	typename
union	unsigned	using	virtual	void
volatile	wchar_t	while		

A titre d'exemple, tous les identificateurs suivants sont corrects :

```
X2    somme_a_payer    Prenom    _Age
```

Alors que les identificateurs suivants ne seront pas acceptés par le compilateur puisqu'ils ne respectent pas les règles précédentes :

```
Prénom    2emevaleur    racine carre    Prix$    this
```

Les noms de variables sont *sensibles à la casse* (le langage C++ fait la différence entre majuscules et minuscules), il faut donc veiller à utiliser des noms comportant la même casse.

La lisibilité des programmes dépend étroitement du choix des noms des variables qui doivent être simples et significatifs. Ainsi, Mont_Fac est un meilleur choix que x pour désigner le montant d'une facture.

2. Déclaration d'une variable

Pour pouvoir utiliser une variable, il faut la définir, c'est-à-dire lui donner un nom, mais surtout un type de données afin qu'un espace mémoire conforme à son type lui soit réservé.

Une variable se déclare de la façon suivante :

type Nom_Variable ;

ou bien s'il y a plusieurs variables du même type :

type Nom_Variable1, Nom_Variable2, ...;

Le langage C++ permet de définir une variable à n'importe quel endroit du code.

Exemple

```
int i;
float x, y;
char c;
```

3. Initialisation d'une variable

La déclaration d'une variable ne fait que « réserver » un emplacement mémoire où stocker la variable. Tant qu'on n'a pas affecté de valeur à la variable, celle-ci contient ce qui se trouvait précédemment à cet emplacement.

En langage C++, on peut affecter une valeur initiale à la variable lors de sa déclaration, on parle alors d'initialisation. La syntaxe à utiliser dans ce cas est la suivante :

type Nom_Variable = donnée;

Exemples

```
int i = 0;
float x = 42.64, y = 166.5;
char c = 'B';
```

4. Affectation d'une valeur à une variable

Pour stocker une donnée dans une variable que l'on a déclarée, il faut faire une affectation, c'est-à-dire préciser la donnée qui va être stockée à l'emplacement mémoire qui a été réservé à cet effet.

Pour cela on utilise l'opérateur d'affectation « = » dont la syntaxe est la suivante:

Nom_Variable = expression;

Exemples

```
float x, y;
x = 3.5;
y = 2 * x + 3;
```

5. Portée (ou visibilité[1]) des variables

Selon l'endroit où l'on déclare une variable, celle-ci pourra être accessible (visible) de partout dans le code ou bien dans une portion limitée de celui-ci (à l'intérieur d'une fonction par exemple), on parle de *portée* (ou *visibilité*) d'une variable.

---

[1] Visibilité = Portée – Partie cachée par un homonyme (variable locale ayant le même nom)

Lorsqu'une variable est déclarée à l'extérieur de toute fonction ou de tout bloc d'instructions, elle est accessible de partout dans le code (n'importe quelle fonction du programme peut faire appel à cette variable), on parle alors de **variable globale**.

Lorsque l'on déclare une variable à l'intérieur d'une fonction ou d'un bloc d'instructions (entre des accolades), sa portée sera limitée à ce seul bloc d'instructions, c'est-à-dire qu'elle est inutilisable ailleurs, on parle alors de **variable locale**.

Pour éviter des effets de bord inattendus, il est préférable d'utiliser des variables locales autant que possible.

Si une variable globale et une variable locale portent le même nom, tout appel à l'intérieur du bloc d'instructions se réfère automatiquement à la variable locale. Pour désigner la variable globale on peut utiliser l'*opérateur de résolution de portée* noté ::. Toutefois, il est préférable d'utiliser des noms de variables différents.

Exemple

```
#include <iostream>
using namespace std;

int i = 3;                    // variable globale

int main( )
{
      int i = 12;             // variable locale
      ::i = i + 8;
      cout<<"Variable globale : "<<::i<<endl;
      cout<<"Variable locale : "<<i<<endl;
      return 0;
}
```

Output du programme

```
Variable globale : 20
Variable locale : 12
```

6. Définition de constantes

Une constante est une donnée dont la valeur est inchangeable lors de l'exécution d'un programme. En langage C++, les constantes sont définies grâce à la directive du préprocesseur *#define*. Par exemple, la directive suivante déclare une constante nommée PI dont la valeur est fixée à 3.1415927 :

```
#define PI 3.1415927
```

Toutefois, avec cette méthode les constantes ne sont pas typées, c'est pourquoi le C++ rajoute le mot réservé *const*, qui permet de définir une constante typée, qui ne pourra pas être modifiée pendant toute la durée d'exécution du programme. Il est ainsi nécessaire d'initialiser la constante dès sa définition.

Exemples

```
const float PI = 3.1415927;
const int TAILLE = 100;
```

Les noms de constantes obéissent aux mêmes règles que les noms de variables. De plus, il est préférable d'utiliser des noms en majuscules pour distinguer les constantes des variables ordinaires.

7. Utilisation des constantes caractères

Il existe des caractères repérés par un code ASCII spécial permettant d'effectuer des opérations particulières à l'écran, l'imprimante ou le beeper de l'ordinateur. Ces caractères peuvent être représentés plus simplement en langage C++ grâce au caractère '\' suivi d'une lettre, qui précise qu'il s'agit d'un caractère de contrôle :

Tableau 3 : Caractères de contrôle en C++

Caractère	Description
\0	caractère de fin de chaîne
\"	guillemet
\\	barre oblique inverse (antislash)
\a	signal sonore (bip)
\b	retour en arrière (backspace)
\f	saut de page (pour l'imprimante)
\n	retour à la ligne
\t	tabulation
\v	tabulation verticale (pour l'imprimante)

En effet, certains de ces caractères ne pourraient pas être représentés autrement (un bip sonore ou un caractère de mise en page ne peuvent pas être représentés à l'écran). D'autre part, les caractères \ et " ne peuvent pas faire partie en tant que tels d'une chaîne de caractères pour des raisons évidentes d'ambiguïté.

Exercice

Identifier parmi la liste suivante les noms de variables valides en C++ :

- Somme
- X2
- π
- 2emeRacine
- Prix-Unitaire
- Montant_Total
- TVA%
- _Solde
- Caractère
- new
- Const

Les opérateurs

1. Qu'est-ce qu'un opérateur ?

Les opérateurs sont des symboles qui permettent de manipuler des variables, c'est-à-dire effectuer des opérations, les évaluer, etc.

On distingue plusieurs types d'opérateurs :

- les opérateurs de calcul
- les opérateurs d'assignation
- les opérateurs d'incrémentation
- les opérateurs de comparaison
- les opérateurs logiques
- les opérateurs bit-à-bit.

2. Les opérateurs de calcul

Les opérateurs arithmétiques classiques sont l'opérateur unaire « - » (changement de signe) ainsi que les opérateurs binaires présentés dans le tableau 4 :

Tableau 4 : Les opérateurs arithmétiques binaires

Opérateur	Dénomination
+	addition
-	soustraction
*	multiplication
/	division
%	modulo (reste de la division)

Exemples

- 6 + 4 / 2 = 8
- (6 + 4) / 2 = 5
- 8 / 4 * 2 = 4
- 8 / (4 * 2) = 1
- 8 % 5 + 6 / (3 - 1) = 6
- 5 * (3 + 12 / 4 * 2 – (5 * (7 – 1) / (2 * 3))) = 20

Remarques

1. Contrairement à d'autres langages, le C++ ne dispose que de la notation « / » pour désigner à la fois la division entière et la division entre flottants. Si les deux opérandes sont de type entier, l'opérateur « / » produira une division entière (quotient de la division). Par contre, il délivrera une valeur flottante dès que l'un des opérandes est un flottant.

   Exemples

   - L'instruction `float x=3/2;` affecte à x la valeur 1.
   - L'instruction `float x=3.0/2;` affecte à x la valeur 1.5.

2. L'opérateur % ne s'applique qu'à des opérandes de type entier. Si l'un des deux opérandes est négatif, le signe du reste dépend de l'implémentation, mais il est en général le même que celui du dividende.

3. En C++, il n'existe pas d'opérateur effectuant l'élévation à la puissance. Il faut utiliser la fonction *pow(x,y)* de la librairie <math.h> pour calculer x^y.

4. Les erreurs fatales telles que la division par zéro ou le calcul de la racine carré d'un nombre négatif peuvent causer l'écroulement du programme.

3. L'opérateur d'affectation (assignation)

Le rôle de cet opérateur consiste à placer une valeur dans une variable. Sa syntaxe générale est de la forme :

Variable = Expression ;

Ainsi, l'instruction :

```
A = 6;
```

signifie « mettre la valeur 6 dans la case mémoire identifiée par A ».

Alors que l'instruction :

```
B = A + 4;
```

range dans B la valeur 10 (A reste toujours égale à 6).

Remarque : La valeur ou le résultat de l'expression à droite du signe d'affectation doit être de même type ou de type compatible avec celui de la variable à gauche. De ce fait, il faut parfois recourir aux opérations de conversion de type vues dans le chapitre précédent (voir page 18).

Exemple : Que vaudront les variables A et B après l'exécution du programme suivant :

```
#include <iostream>
using namespace std;
int main( )
{
        int A = 5, B = 7;
        A = A + B;
        B = A - B;
        A = A - B;
        cout<<"A = "<<A<<endl;
        cout<<"B = "<<B<<endl;
        return 0;
}
```

Output du programme

```
A = 7
B = 5
```

Ce programme fait la permutation des valeurs de A et B.

Le programme suivant fait également la permutation des valeurs de A et B mais en utilisant une variable intermédiaire temp :

```
#include <iostream>
using namespace std;
int main( )
{
        int A = 5, B = 7, temp;
        temp = A;
        A = B;
        B = temp;
        cout<<"A = "<<A<<endl;
        cout<<"B = "<<B<<endl;
        return 0;
}
```

Exercice

Soient A, B et C trois variables de type entier. Ecrire un programme qui fait une permutation circulaire de sorte que la valeur de A passe dans B, celle de B passe dans C et celle de C passe dans A. On utilisera une seule variable supplémentaire.

4. Les opérateurs d'affectation composée

Ces opérateurs permettent de simplifier des opérations telles que *ajouter la valeur 5 à la variable x et stocker le résultat dans la même variable x*. Une telle opération s'écrirait habituellement de la façon suivante :

```
x = x + 5;
```

Avec les opérateurs d'affectation composée, il est possible d'écrire cette opération sous la forme suivante:

```
x += 5;
```

Ainsi, si la valeur de x était 7 avant opération, elle sera 12 après.

Le tableau 5 ci-dessous présente une brève description des principaux opérateurs composés disponibles en C++ :

Tableau 5 : Les opérateurs d'affectation composée

Opérateur	Effet
+=	additionne deux valeurs et stocke le résultat dans la variable à gauche
-=	soustrait deux valeurs et stocke le résultat dans la variable à gauche
*=	multiplie deux valeurs et stocke le résultat dans la variable à gauche
/=	divise deux valeurs et stocke le résultat dans la variable à gauche
%=	calcule le reste de la division de deux valeurs et stocke le résultat dans la variable à gauche

5. Les opérateurs d'incrémentation

Ce type d'opérateur permet d'augmenter ou diminuer facilement la valeur d'une variable d'une unité. Ces opérateurs sont très utiles pour des structures telles que les boucles, qui ont besoin d'un compteur (variable qui augmente de un en un).

Un opérateur de type **x++** permet de remplacer des notations lourdes telles que x=x+1 ou bien x+=1.

Tableau 6 : Les opérateurs d'incrémentation

Opérateur	Dénomination	Effet	Syntaxe	Résultat (avec x valant 7)
++	incrémentation	augmente la variable d'une unité	x++	8
--	décrémentation	diminue la variable d'une unité	x--	6

Les opérateurs d'incrémentation (++) et de décrémentation (--) s'appliquent comme des ***préfixes*** ou des ***suffixes*** sur les variables. Lorsqu'ils sont en préfixe, la variable est incrémentée ou décrémentée, puis sa valeur est renvoyée. S'ils sont en suffixe, la valeur de la variable est renvoyée, puis la variable est incrémentée ou décrémentée.

Exemple

```
#include <iostream>
using namespace std;

int main( )
{
        int i = 2, j, k;
        j = ++i;
        k = j++;
        cout<<"i = "<<i<<endl;
        cout<<"j = "<<j<<endl;
        cout<<"k = "<<k<<endl;
        return 0;
}
```

Output du programme

```
i = 3
j = 4
k = 3
```

6. Les opérateurs de comparaison

Le tableau 7 montre les différents opérateurs de comparaison disponibles en C++ :

Tableau 7 : Les opérateurs de comparaison

Opérateur	Dénomination	Effet	Exemple	Résultat
==	opérateur d'égalité	compare deux valeurs et vérifie leur égalité	x==3	retourne 1 si x est égal à 3, sinon 0
<	opérateur d'infériorité stricte	vérifie qu'une variable est strictement inférieure à une valeur	x<3	retourne 1 si x est inférieur à 3, sinon 0
<=	opérateur d'infériorité large	vérifie qu'une variable est inférieure ou égale à une valeur	x<=3	retourne 1 si x est inférieur ou égal à 3, sinon 0
>	opérateur de supériorité stricte	vérifie qu'une variable est strictement supérieure à une valeur	x>3	retourne 1 si x est supérieur à 3, sinon 0
>=	opérateur de supériorité large	vérifie qu'une variable est supérieure ou égale à une valeur	x>=3	retourne 1 si x est supérieur ou égal à 3, sinon 0
!=	opérateur de différence	vérifie qu'une variable est différente d'une valeur	x!=3	retourne 1 si x est différent de 3, sinon 0

Exemples : Si on pose x = 5 et y = 2

- (x == 5) = true
- (x <= (y + 2)) = false
- (x != (y + 3)) = false

7. Les opérateurs logiques (booléens)

Ces opérateurs permettent de vérifier si plusieurs conditions sont vraies :

Tableau 8 : Les opérateurs logiques

<table>
<tr><th>Opérateur</th><th>Dénomination</th><th>Effet</th><th>Syntaxe</th></tr>
<tr><td>||</td><td>OU logique</td><td>vérifie qu'une des conditions est réalisée</td><td>(condition1)||(condition2)</td></tr>
<tr><td>&&</td><td>ET logique</td><td>vérifie que toutes les conditions sont réalisées</td><td>(condition1)&&(condition2)</td></tr>
<tr><td>!</td><td>NON logique</td><td>inverse l'état d'une variable booléenne (retourne la valeur 1 si la variable vaut 0, 0 si elle vaut 1)</td><td>(!condition)</td></tr>
</table>

La table de vérité suivante montre les différents cas de figure de combinaison de deux expressions logiques A et B:

Tableau 9 : Table de vérité des opérateurs logiques

<table>
<tr><th>A</th><th>B</th><th>A && B</th><th>A || B</th><th>! A</th></tr>
<tr><td>true</td><td>true</td><td>true</td><td>true</td><td>false</td></tr>
<tr><td>true</td><td>false</td><td>false</td><td>true</td><td>false</td></tr>
<tr><td>false</td><td>true</td><td>false</td><td>true</td><td>true</td></tr>
<tr><td>false</td><td>false</td><td>false</td><td>false</td><td>true</td></tr>
</table>

Exemples : Si on pose x = 5, y = 2 et z = 3

- (x > y + z) && (y % 2 == 0) = false
- (y + z <= x) || (z < 0) = true
- !(z != y + 1) = true

Remarque : En C++, la valeur *true* est représentée par 1 alors que la valeur *false* est représentée par 0. A titre d'exemple, l'instruction `cout<<(8>5)` ; affiche 1 à l'écran.

8. Les opérateurs bit- à -bit

Ces opérateurs traitent les données selon leur représentation binaire mais retournent des valeurs numériques standards dans le format d'origine. Les opérations sont effectuées bit-à-bit, c'est-à-dire avec les bits de même poids.

Tableau 10 : Les opérateurs bit- à -bit

<table>
<tr><th>Opérateur</th><th>Dénomination</th><th>Effet</th><th>Syntaxe</th><th>Résultat</th></tr>
<tr><td>&</td><td>ET bit-à-bit</td><td>retourne 1 si les deux bits de même poids sont à 1</td><td>9 & 12
(1001 & 1100)</td><td>8
(1000)</td></tr>
<tr><td>|</td><td>OU bit-à-bit</td><td>retourne 1 si l'un ou l'autre des deux bits de même poids est à 1 (ou les deux)</td><td>9 | 12
(1001 | 1100)</td><td>13
(1101)</td></tr>
<tr><td>^</td><td>OU exclusif bit-à-bit</td><td>retourne 1 si l'un des 2 bits de même poids est à 1 (mais pas les deux)</td><td>9 ^ 12
(1001 ^ 1100)</td><td>5
(0101)</td></tr>
<tr><td>~</td><td>NON binaire</td><td>retourne le complément à deux[2] du nombre</td><td>~6</td><td>-7</td></tr>
</table>

[2] Le complément à 2 est une méthode de représentation des nombres signés. Les nombres positifs sont représentés en base 2. Pour les nombres négatifs, on part de la valeur absolue que l'on représente en binaire, on la complémente bit à bit et on additionne 1. A Titre d'exemple, sur 8 bits, la représentation la valeur 6 est 00000110 alors que la représentation de la valeur -6 est 11111010.

9. Les opérateurs de décalage de bit

Les opérateurs suivants décalent chacun des bits d'un nombre de bits vers la gauche ou vers la droite. Le premier opérande désigne le nombre sur lequel on va faire le décalage, le second désigne le nombre de bits duquel il va être décalé.

Tableau 11 : Les opérateurs de décalage de bit

Opérateur	Dénomination	Effet	Syntaxe	Résultat
<<	décalage à gauche	décale les bits vers la gauche (multiplie par 2 à chaque décalage). Les zéros qui sortent à gauche sont perdus, tandis que des zéros sont insérés à droite.	6 << 1 (110 << 1)	12 (1100)
>>	décalage à droite avec conservation du signe	décale les bits vers la droite (divise par 2 à chaque décalage). Les zéros qui sortent à droite sont perdus, tandis que le bit non nul de poids plus fort est recopié à gauche.	6 >> 1 (0110 >> 1)	3 (0011)

Exemple

```
#include <iostream>
using namespace std;

int main( )
{
        int a = 10;
        cout<<"a = "<<a<<endl;
        cout<<"a << 1 = "<<(a<<1)<<endl;
        cout<<"a >> 1 = "<<(a>>1)<<endl;
        return 0;
}
```

Output du programme

```
a = 10
a << 1 = 20
a >> 1 = 5
```

10. Les priorités

Lorsque l'on associe plusieurs opérateurs, il faut que le compilateur sache dans quel ordre les traiter (ou plus exactement dans quel ordre il doit exécuter les opérations), voici donc dans **l'ordre décroissant** les priorités de tous les opérateurs :

Tableau 12 : Priorité des opérateurs

<table>
<tr><th>Niveau de priorité</th><th>Opérateurs</th></tr>
<tr><td>1</td><td>( ), [ ]</td></tr>
<tr><td>2</td><td>++, --, !, ~, - (signe négatif)</td></tr>
<tr><td>3</td><td>*, /, %</td></tr>
<tr><td>4</td><td>+, - (soustraction)</td></tr>
<tr><td>5</td><td><, <=, >, >=</td></tr>
<tr><td>6</td><td>==, !=</td></tr>
<tr><td>7</td><td>^</td></tr>
<tr><td>8</td><td>|</td></tr>
<tr><td>9</td><td>&&, ||</td></tr>
<tr><td>10</td><td>?, :</td></tr>
<tr><td>11</td><td>=, +=, -=. *=, /=, %=, <<=, >>=, >>>=, &=, ^=, |=</td></tr>
<tr><td>12</td><td>,</td></tr>
</table>

Exercice 1

1. Evaluer les expressions arithmétiques suivantes :

   - 8 / 4 * 2 + 6 / 2 + 5
   - -5 % 3
   - 13 / 2 % 2
   - 8 % 5 + 4 / (5 – 2)
   - 5 + 2 * 6 - 4 + (8 + 2 % 3) / (2 - 4 + 5 * 2)

2. Evaluer les expressions logiques suivantes (on pose a=4, b=5, c=-1 et d=0) :

   - (a < b) && (c >= d)
   - !(a < b) || (c != b)
   - !((a != 2 * b) || (a * c < d))

Exercice 2

Ecrire un programme en C++ qui lit la longueur et la largeur d'un terrain rectangulaire (en m) puis calcule et affiche sa surface (en m^2).

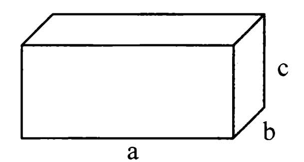
Exercice 3

Ecrire un programme en C++ qui lit le rayon d'un cercle (en cm) et qui affiche ensuite son périmètre et sa surface en utilisant les formules suivantes :

- $P = 2 \times \pi \times R$
- $S = \pi \times R^2$

Exercice 4

Ecrire un programme qui calcule le volume d'un réservoir qui a la forme d'un parallélépipède rectangle de longueur a, de largeur b et de hauteur c.



Exercice 5

Ecrire un programme qui calcule et affiche la résistance d'un composant électrique en utilisant la loi d'Ohm :

$$U = R \times I \qquad \text{avec} \begin{cases} U : \text{Tension en V} \\ R : \text{Résistance en } \Omega \\ I : \text{Intensité en A.} \end{cases}$$

Exercice 6

Ecrire un programme en C++ qui calcule le salaire net d'un employé en utilisant la formule suivante :

Salaire net = Salaire brut – Impôt

Sachant que l'impôt est fixé à 15% du salaire brut.

Exemples

Salaire brut (en TND)	Impôt (en TND)	Salaire net (en TND)
400	60	340
1000	150	850

Exercice 7

Ecrire un programme en C++ qui transforme un temps exprimé en secondes et un temps équivalent exprimé en heures, minutes et secondes.

Exemples

Entrée (Secondes)	Sortie		
	Heures	Minutes	Secondes
50	0	0	50
100	0	1	40
500	0	8	20
4000	1	6	40

Exercice 8

1. Ecrire un programme qui simule un guichet automatique de billets.

Le programme demande au client de saisir le montant à retirer (multiple de 10) puis renvoie le nombre de billets de 10, 20 et 30 dinars à remettre au client. On suppose toujours que le solde disponible est suffisant et que l'on désire offrir au client un nombre minimum de billets.

Exemples

Montant demandé	Nbre de billets de 30 D	Nbre de billets de 20 D	Nbre de billets de 10 D
10	0	0	1
20	0	1	0
30	1	0	0
40	1	0	1
50	1	1	0
100	3	0	1
200	6	1	0

2. Modifier votre programme pour prendre en compte les billets de 50 dinars.

Les structures de contrôle

1. La notion de bloc

Une expression suivie d'un point-virgule est appelée ***instruction***. Par exemple `a++;` est une instruction. Lorsque l'on veut regrouper plusieurs instructions, on crée ce qu'on appelle un ***bloc***, c'est-à-dire un ensemble d'instructions (suivies respectivement par des points-virgules) et comprises entre les accolades { et }.

Les instructions *if*, *for* et *while* peuvent par exemple être suivies d'un bloc d'instructions à exécuter.

2. L'instruction if

L'instruction *if* est la structure de test la plus basique, on la retrouve dans tous les langages de programmation (avec une syntaxe différente). Elle permet d'exécuter une série d'instructions lorsqu'une condition est vérifiée.

La syntaxe de cette expression est la suivante :

```
if (condition)
{
        liste d'instructions
}
```

L'exécution de cette instruction se déroule selon le schéma de la figure 2:

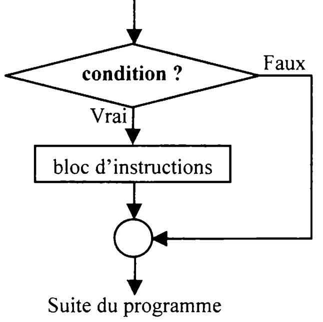


Figure 2 : Schéma d'exécution de l'instruction if

Remarques

1. la condition doit être toujours mise entre deux parenthèses.
2. il est possible de définir plusieurs conditions à remplir en utilisant les opérateurs && (ET) et || (OU).
3. s'il n'y a qu'une instruction à exécuter, les accolades ne sont pas nécessaires.

Exercice

Ecrire un programme qui calcule le salaire hebdomadaire d'un ouvrier à partir du nombre d'heures travaillées, du taux horaire et du nombre d'années de service (ancienneté). Les ouvriers ayant une ancienneté de plus de 10 ans ont le droit à une prime supplémentaire de 50 D.

Solution

```
#include <iostream>
using namespace std;

int main( )
{
       float nbh, th, anciennete;
       float salaire;
       cout<<"Entrer le nbre d'heures travaillees : ";
       cin>>nbh;
       cout<<"Entrer le taux horaire : ";
       cin>>th;
       cout<<"Entrer l'anciennete : ";
       cin>>anciennete;
       salaire = nbh * th;
       if (anciennete > 10)
              salaire = salaire + 50;
       cout<<"Salaire = "<<salaire<<" TND"<<endl;
       return 0;
}
```

Test du programme

```
Entrer le nbre d'heures travaillees : 40
Entrer le taux horaire : 2.5
Entrer l'anciennete : 11
Salaire = 150 TND
```

3. L'instruction if ... else

L'expression *if ... else* permet d'exécuter une série d'instructions si une condition est réalisée et une autre série d'instructions en cas de non réalisation de cette condition.

La syntaxe de cette expression est la suivante :

```
if (condition)
{
        liste d'instructions
}
else
{
        autre liste d'instructions
}
```

L'exécution de cette instruction se déroule selon le schéma de la figure 3:

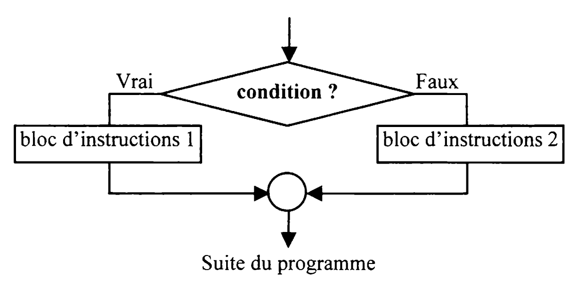


Figure 3 : Schéma d'exécution de l'instruction if ... else

Exemple : Ecrire un programme qui calcule la valeur absolue d'un réel x en utilisant la formule suivante:

$$Abs(x) = |x| = \begin{cases} x & si\, x \geq 0 \\ -x & si\, x < 0 \end{cases}$$

Solution

```
#include <iostream>
using namespace std;

int main( )
{
      float x, abs;
      cout<<"Entrer un nombre : ";
      cin>>x;
      if (x >= 0)
            abs = x;
      else
            abs = -x;
      cout<<"abs = "<<abs<<endl;
      return 0;
}
```

Test du programme

```
Entrer un nombre : -13.5
abs = 13.5
```

Exercice

Ecrire un programme qui lit un entier n puis vérifie si ce nombre est pair ou impair.

Solution

```
#include <iostream>
using namespace std;

int main( )
{
      int n;
      cout<<"Entrer un entier : ";
      cin>>n;
      if (n % 2 == 0)
            cout<<"C'est un nombre pair";
      else
            cout<<"C'est un nombre impair";
      cout<<endl ;
      return 0;
}
```

Remarque

Plusieurs instructions *if* peuvent être imbriquées les unes dans les autres. Le *else* se rapporte toujours au dernier *if* rencontré auquel un *else* n'est pas encore attribué.

Exemple

Ecrire un programme qui lit le montant brut d'une facture puis calcule le montant net à payer après déduction d'une remise dont le taux varie selon le tableau suivant :

Montant de la facture (TND)	Taux de la remise (%)
< 1000	2
< 10000	5
< 50000	7
≥ 50000	10

Solution

```
#include <iostream>
using namespace std;

int main( )
{
    float montant, tx_remise, mont_remise,
net_a_payer;
    cout<<"Entrer le montant de la facture: ";
    cin>>montant;
    if (montant < 1000) tx_remise = 2;
    else if (montant < 10000) tx_remise = 5;
    else if (montant < 50000) tx_remise = 7;
    else tx_remise = 10;
    mont_remise = montant * tx_remise / 100 ;
    net_a_payer = montant - mont_remise;
    cout<<"Taux de remise="<<tx_remise<<"%"<<endl;
    cout<<"Net A Payer = "<<net_a_payer<<endl;
    return 0;
}
```

Test du programme

```
Entrer le montant de la facture: 8000
Taux de remise = 5 %
Net A Payer = 7600
```

4. L'opérateur conditionnel (?)

En C++, Il est possible de faire un test avec une structure beaucoup moins lourde, appelée opérateur conditionnel, dont la syntaxe est la suivante :

(condition) ? instruction si vrai : instruction si faux ;

Exemple

```
#include <iostream>
using namespace std;

int main( )
{
        float x, abs;
        cout<<"Entrer un nombre : ";
        cin>>x;
        abs = (x >= 0) ? x : -x;
        cout<<"abs = "<<abs<<endl;
        return 0;
}
```

Exercice 1

1. Ecrire un programme qui lit deux entiers puis affiche leur minimum en utilisant la structure if.
2. Réécrire le programme précédent en utilisant l'opérateur conditionnel (?).

Exercice 2 : Ecrire un programme qui lit trois entiers puis affiche leur maximum.

Exercice 3 : Ecrire un programme qui calcule le montant à payer à la STEG contre la consommation d'électricité pendant une période sachant que les tarifs de consommation sont fixés comme suit :

Tranche	Prix unitaire
0 .. 200 KW	93 Millimes
> 200 KW	134 Millimes

<u>Exemples</u>

Consommation (KW)	Montant à payer (Millimes)
100	9300
300	32000

5. L'instruction switch

L'instruction *switch* permet de faire plusieurs tests de valeurs sur le contenu d'une même variable. Ce branchement conditionnel simplifie beaucoup le test, car cette opération aurait été compliquée (mais possible) avec des instructions *if* imbriquées. Sa syntaxe est la suivante :

```
switch (expression)
{
        case valeur1 :
                liste d'instructions
                break;
        case valeur2 :
                liste d'instructions
                break;
        ...
        default:
                liste d'instructions
                break;
}
```

Les parenthèses qui suivent le mot clé *switch* indiquent une expression dont la valeur est testée successivement par chacun des *case*. Lorsque l'expression testée est égale à l'une des valeurs suivant un *case*, la liste d'instructions qui suit celui-ci est exécutée. Le mot clé *break* indique la sortie de la structure conditionnelle. Le mot clé *default* précède la liste d'instructions qui sera exécutée si l'expression n'est égale à aucune des valeurs mentionnées.

⚠ Il ne faut pas oublier d'insérer des instructions *break* à la fin de chaque test, ce genre d'oubli est difficile à détecter car aucune erreur ne sera signalée par le compilateur.

Exemple

Ecrire un programme qui lit un numéro entre 1 et 12 puis affiche le nom du mois correspondant. Si la valeur entrée est en dehors de cet intervalle, le programme doit afficher un message d'erreur.

Solution

```
#include <iostream>
using namespace std;

int main()
{
	int mois;
	cout<<"Enter le numero du mois : ";
	cin>>mois;
	switch (mois)
	{
		case 1:  cout<<"Janvier"<<endl ;
		         break;
		case 2:  cout<<"Fevrier"<<endl;
		         break;
		case 3:  cout<<"Mars"<<endl;
		         break;
		case 4:  cout<<"Avril"<<endl;
		         break;
		case 5:  cout<<"Mai"<<endl;
		         break;
		case 6:  cout<<"Juin"<<endl;
		         break;
		case 7:  cout<<"Juillet"<<endl;
		         break;
		case 8:  cout<<"Aout"<<endl;
		         break;
		case 9:  cout<<"Septembre"<<endl;
		         break;
		case 10: cout<<"Octobre"<<endl;
		         break;
		case 11: cout<<"Novembre"<<endl;
		         break;
		case 12: cout<<"Decembre"<<endl;
		         break;
		default: cout<<"Mois inexistant ...";
		         break;
	}
	cout<<endl;
	return 0;
}
```

Test du programme

```
Enter le numero du mois : 8
Aout
```

Exercice

Ecrire un programme qui simule une calculatrice à 4 opérations (+, -, *, /). L'opération à effectuer doit être introduite dans le format A op B.

Exemples : 8+6, 14-8, 3.5*4, 14/3.5

Utiliser la structure *switch* pour choisir l'opération à effectuer.

Solution

```
#include <iostream>

using namespace std;

int main()
{
	float A, B;
	char Operation;
	cout<<"Enter votre operation : ";
	cin>>A>>Operation>>B;
	switch (Operation)
	{
		case '+':
			cout<<"Resultat = "<<A+B;
			break;
		case '-':
			cout<<"Resultat = "<<A-B;
			break;
		case '*':
			cout<<"Resultat = "<<A*B;
			break;
		case '/':
			if (B != 0)
			    cout<<"Resultat = "<<A/B;
			else
			    cout<<"Division par zero !";
			break;
```

```
                default:
                        cout<<"Operateur invalide ...";
                        break;
        }
        cout<<endl;
        return 0;
}
```

Test du programme

```
Enter votre operation : 150/6
Resultat = 25
```

6. Les boucles

Les boucles sont des structures qui permettent d'exécuter plusieurs fois la même série d'instructions jusqu'à ce qu'une condition ne soit plus réalisée.

On appelle parfois ces structures *instructions répétitives* ou *itérations*. La façon la plus commune de faire une boucle, est de créer un compteur (une variable qui s'incrémente, c'est-à-dire qui augmente de 1 à chaque tour de boucle) et de faire arrêter la boucle lorsque le compteur dépasse une certaine limite.

a. La boucle for

La structure de contrôle *for* est sans doute l'une des plus importantes. Elle permet de réaliser toutes sortes de boucles et, en particulier, les boucles itérant sur les valeurs d'une variable de contrôle (compteur).

La syntaxe de la boucle *for* est la suivante :

```
for (initialisation ; condition ; itération)
{
        liste d'instructions
}
```

où :

- *initialisation* est une instruction (ou un bloc d'instructions séparées par des virgules) exécutée une seule fois avant le premier parcours de la boucle ;

- *condition* est une expression logique dont la valeur déterminera la fin de la boucle ;
- *itération* est l'opération à effectuer à la fin de chaque itération pour changer la valeur du compteur.

Toutes ces parties sont facultatives.

L'exécution de cette instruction se déroule selon le schéma de la figure 4 :

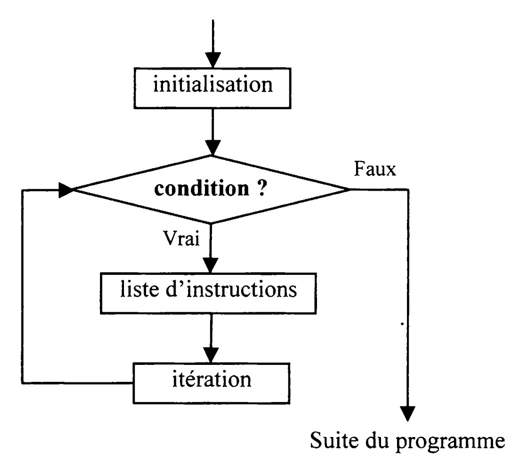


Figure 4 : Schéma d'exécution de la boucle for

Exemple

Ecrire un programme qui calcule et affiche la somme des entiers de 1 à 50.

Solution

```
#include <iostream>

using namespace std;

int main()
{
        int i;
        int somme = 0;
        for (i = 1; (i <= 50); i++)
                somme = somme + i;
        cout<<"Somme = "<<somme<<endl;
        return 0;
}
```

Output du programme

```
Somme = 1275
```

Remarques

1. il faut toujours vérifier que la boucle a bien une condition de sortie (i.e. le compteur s'incrémente correctement).

2. il faut bien compter le nombre de fois que l'on veut faire exécuter la boucle :

   - `for(i=0; i<10; i++)` exécute 10 fois la boucle (i de 0 à 9)
   - `for(i=0; i<=10; i++)` exécute 11 fois la boucle (i de 0 à 10)
   - `for(i=1; i<10; i++)` exécute 9 fois la boucle (i de 1 à 9)
   - `for(i=1; i<=10; i++)` exécute 10 fois la boucle (i de 1 à 10)

3. si le corps de la boucle ne comporte qu'une seule instruction, les accolades ne sont pas nécessaires.

Exercice 1

Ecrire un programme qui calcule la somme harmonique $s = \sum_{i=1}^{n} \frac{1}{i}$; n est un entier positif lu à partir du clavier.

<u>Exemple :</u> Pour n = 3, s = 1 + 1/2 + 1/3 = 1.83.

Exercice 2

Ecrire un programme en C++ qui affiche tous les entiers impairs compris entre 5 et 55.

Exercice 3

Ecrire un programme en C++ qui lit un entier positif n puis affiche tous ses diviseurs à l'écran.

A titre d'exemple, les diviseurs de 30 sont 1, 2, 3, 5, 6, 10, 15, 30.

Exercice 4

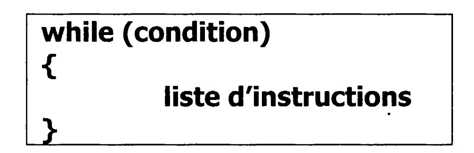
1. Ecrire un programme en C++ qui lit les notes de 10 étudiants puis calcule et affiche la note moyenne.

2. Modifier le programme pour qu'il fournisse également la meilleure et la mauvaise note.

b. La boucle while

La structure *while* permet d'exécuter les instructions d'une boucle tant qu'une condition est vérifiée.

Sa syntaxe est la suivante :


```
while (condition)
{
        liste d'instructions
}
```

L'exécution de cette instruction se déroule selon le schéma de la figure 5:

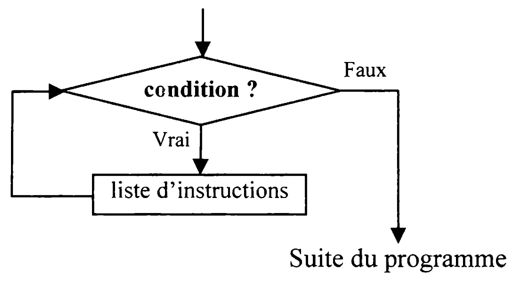


Figure 5 : Schéma d'exécution de la boucle while

Les risques de **boucle infinie** (boucle dont la condition est toujours vraie) sont grands, ceci risque de provoquer une erreur système par manque de ressources. De ce fait, il faut toujours s'assurer que la condition de sortie sera vérifiée après un nombre fini de parcours.

Exemple : Ecrire un programme qui calcule et affiche la somme des entiers de 1 à 50 en utilisant une boucle while.

```
#include <iostream>
using namespace std;

int main()
{
	int i;
	int somme = 0;
	i = 1;
	while (i <= 50)
	{
		somme = somme + i;
		i++;
	}
	cout<<"Somme = "<<somme<<endl;
	return 0;
}
```

c. Transition entre la boucle for et la boucle while

D'une façon générale, le passage d'une boucle *for* à une boucle *while* ou l'inverse peut se faire conformément au schéma suivant :

```
for (init; condition; itération)
     opération;
```

⇔

```
init;
while (condition)
{
    opération;
    itération;
}
```

Exercice 1 : Ecrire un programme qui calcule factoriel d'un entier positif n en utilisant la boucle *for* puis la boucle while.

Le factoriel d'un entier n se calcule à partir de la formule suivante:

$$n\,! = n \times (n-1) \times (n-2) \times \ldots \times 2$$

Sachant que 0 ! = 1 et 1 ! = 1.

<u>Exemples :</u> 3 ! = 6 5 ! = 120 6 ! = 720

Solution 1 (for)	Solution 2 (while)

`#include <iostream>`
 	
`using namespace std;`
 	
`int main( )`
`{`
`   int  i, n, f;`
`   cout<<"Entrer n : ";`
`   cin>>n;`
`   f = 1;`
`   for (i = n; i >= 2; i--)`
`   {`
`       f = f * i;`
`   }`
`   cout<<n<<"! = "<<f<<endl;`
`   return 0;`
`}`
	`#include <iostream>`
	
	`using namespace std;`
	
	`int main( )`
	`{`
	`   int  i, n, f;`
	`   cout<<"Entrer n: ";`
	`   cin>>n;`
	`   f = 1;`
	`   i = n;`
	`   while (i >= 2)`
	`   {`
	`       f = f * i;`
	`       i--;`
	`   }`
	`   cout<<n<<"! = "<<f<<endl;`
	`   return 0;`
	`}`

Exercice 2

Ecrire un programme qui lit deux entiers a et b puis calcule et affiche leur PGCD (Plus Grand Commun Diviseur) en utilisant la méthode suivante :

- PGCD (a,b) = a si (a = b)
- PGCD (a,b) = PGCD(a-b, b) si (a > b)
- PGCD (a,b) = PGCD(a, b-a) si (b > a)

Exemples

- PGCD(30,20) = PGCD(10,20) = PGCD(10,10) = 10
- PGCD(18,45) = PGCD(18,27) = PGCD(18,9) = PGCD(9,9) = 9

Solution

```
#include <iostream>
using namespace std;

int main()
{
      int a, b;
      cout<<"Enter le premier entier : ";
      cin>>a;
```

```
        cout<<"Enter le deuxieme entier: ";
        cin>>b;
        while (a != b)
        {
                if (a > b)
                        a = a - b;
                else
                        b = b - a;
        }
        cout<<"PGCD = "<<a<<endl;
        return 0;
}
```

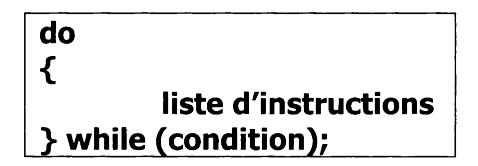
Test du programme

```
Enter le premier entier : 21
Enter le deuxieme entier: 35
PGCD = 7
```

d. La boucle do ... while

La structure de contrôle *do ... while* permet, tout comme la structure *while*, de réaliser des boucles en attente d'une condition.

La syntaxe de cette boucle est la suivante :



```
do
{
        liste d'instructions
} while (condition);
```

L'exécution de cette instruction se déroule selon le schéma de la figure 6 :

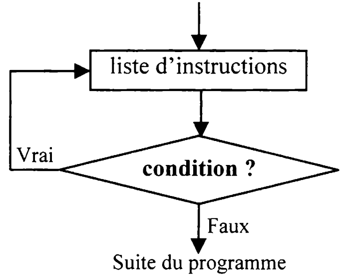


Figure 6 : Schéma d'exécution de la boucle do ... while

Contrairement à la boucle *while*, la boucle *do … while* effectue le test sur la condition après l'exécution des instructions. Cela signifie que les instructions sont toujours exécutées au moins une fois, que la condition soit vérifiée ou non.

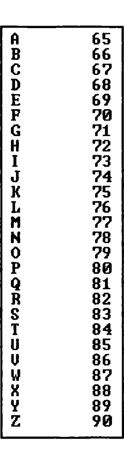
Exemple : Ecrire un programme qui affiche les lettres de l'alphabet majuscules avec leurs codes ASCII en utilisant une boucle *do … while*.

```
#include <iostream>

using namespace std;

int main()
{
	char c = 'A';
	do
	{
		cout<<c<<"\t"<<(int)c<<endl;
		c = c + 1;
	} while (c <= 'Z');
	cout<<endl;
	return 0;
}
```

Output du programme



```
A	65
B	66
C	67
D	68
E	69
F	70
G	71
H	72
I	73
J	74
K	75
L	76
M	77
N	78
O	79
P	80
Q	81
R	82
S	83
T	84
U	85
V	86
W	87
X	88
Y	89
Z	90
```

e. Les boucles imbriquées

Le corps d'une boucle (*for*, *while* ou *do … while*) peut comporter une autre boucle mais avec un compteur différent.

Exemple

```
#include <iostream>

using namespace std;

int main()
{
	int i, j;
	for (i = 1; i <= 10; i++)
	{
		for (j = 1; j <= 10; j++)
			cout<<i*j<<"\t";
		cout<<endl;
	}
	return 0;
}
```

Output du programme

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

7. Saut inconditionnel

Il peut être nécessaire de faire sauter à la boucle une ou plusieurs valeurs sans pour autant mettre fin à celle-ci. Pour ce faire, on utilise l'instruction *continue* généralement associée à une structure conditionnelle, sinon les lignes situées entre cette instruction et la fin de la boucle seraient obsolètes.

Exemple : Imaginons que l'on veuille imprimer pour x allant de 1 à 10 la valeur de 1/(x-7) ; il est évident que pour (x=7) il y aura une erreur. Grâce à l'instruction *continue* il est possible de traiter cette valeur à part puis continuer la boucle.

```
#include <iostream>

using namespace std;

int main()
{
	for (int x = 1; x <= 10; x++)
	{
		if (x == 7)
		{
			cout<<"Division par zero …"<<endl;
			continue;
		}
		cout<<1.0/(x-7)<<endl;
	}
	return 0;
}
```

Output du programme

```
-0.166667
-0.2
-0.25
-0.333333
-0.5
-1
Division par zero ...
1
0.5
0.333333
```

8. Arrêt inconditionnel

L'instruction *break* permet d'arrêter prématurément une boucle (*for*, *while* ou *do..while*). Il s'agit, tout comme *continue*, de l'associer à une structure conditionnelle, sinon la boucle ne fera jamais plus d'un tour.

Exemple

```
#include <iostream>
using namespace std;

int main()
{
	for (int x = 1; x <= 10; x++)
	{
		if (x == 7)
		{
			cout<<"Division par zero …"<<endl;
			break;
		}
		cout<<1.0/(x-7)<<endl;
	}
	return 0;
}
```

Output du programme

```
-0.166667
-0.2
-0.25
-0.333333
-0.5
-1
Division par zero ...
```

Exercice

Il existe un seul nombre entier appartenant à l'intervalle [1..500] et qui vérifie les propriétés suivantes :

x mod 2 = 1
x mod 3 = 1
x mod 4 = 1
x mod 5 = 1
x mod 6 = 1
x mod 7 = 0

Ecrire un programme qui permet de trouver ce nombre x.

Solution

```
#include <iostream>
using namespace std;

int main()
{
   int i;
   for (i = 1; i <= 500; i++)
      if ((i%2==1)&&(i%3==1)&&(i%4==1)&&(i%5==1)&&
      (i%6==1)&&(i%7==0))
      {
         cout<<"le nombre recherche est "<<i;
         break;
      }
   cout<<endl;
   return 0;
}
```

Output du programme

```
Le nombre recherche est 301
```

Exercice (QCM) : Cocher à chaque fois la bonne réponse

1- Que signifie le symbole <= ?

- O Supérieur ou égal
- O Inférieur ou égal
- O Différent de

2- Si la variable TempsRestant vaut 0, qu'affiche ce code ?

```
if (TempsRestant != 0)
    cout<<"Il vous reste du temps !";
else
    cout<<"C'est fini !";
```

- O Il vous reste du temps !
- O C'est fini !

3- Comment faire en sorte que la variable *Continuer* soit un booléen valant vrai s'il reste encore du temps dans la variable TempsRestant ?

- O `Continuer = (TempsRestant > 1);`
- O `Continuer = (TempsRestant != 1);`
- O `Continuer = (TempsRestant > 0);`
- O `Continuer = (TempsRestant >= 0);`

4- Qu'affichera ce code ?

```
bool riche, majeur;
int ArgentEnPoche = 1000, age = 19;
riche = (ArgentEnPoche > 1000);
majeur = !(age < 18);
if (riche && majeur)
    cout<<"Vous pouvez ouvrir un compte en banque";
else
    cout<<"Vous ne pouvez pas ouvrir un compte";
```

- O Vous pouvez ouvrir un compte en banque
- O Vous ne pouvez pas ouvrir un compte

5- Quel est le problème de ce *switch* ?

```
switch (variable)
{
   case 1:
      cout<<"Bonjour";
   case 2:
      cout<<"Bonsoir";
   default:
      cout<<"Au revoir";
}
```

- O Il faut mettre un point-virgule à la fin du *switch*
- O Il faut ouvrir des accolades pour chaque *case*
- O Il manque des instructions *break*
- O Il faut écrire *case default* et non pas *default* tout court

6- Que vaudra la variable n après ce code ?

```
int nbTouches = 108 ;
int sansFil = 0 ;
int n = (sansFil || nbTouches >= 108) ? 20 : 30;
n = (n==20 && (sansFil && nbTouches >= 108)) ? 40 : 50;
```

- O 20
- O 30
- O 40
- O 50

7- Laquelle de ces boucles n'existe pas en C++ ?

- O *for*
- O *repeat*
- O *do.. while*
- O *while*

8- Combien de fois le message "Salut" sera-t-il affiché ?

```
int compteur = 15;
do
{
    cout<<"Salut\n";
    compteur++;
} while (compteur < 15);
```

- O 0 fois
- O 1 fois
- O 15 fois
- O 16 fois

9- Combien de fois le message "Salut" sera-t-il affiché ici ?

```
int compteur = 14;
while (compteur < 15)
{
    cout<<"Salut\n";
}
```

- O 0 fois
- O 1 fois
- O 15 fois
- O C'est une boucle infinie

10- Laquelle de ces boucles *for* pourrait afficher les messages suivants dans la console :

```
Ligne No 1
Ligne No 3
Ligne No 5
Ligne No 7
```

O
```
for (int i = 1 ; i < 9 ; i += 2)
    cout<<"Ligne No "<<i<<endl ;
```

O
```
for (int i = 1 ; i <= 7 ; i++)
    cout<<"Ligne No "<<i<<endl ;
```

O
```
for (int i = 0 ; i < 9 ; i += 2)
    cout<<"Ligne No "<<i<<endl ;
```

EXERCICES D'APPLICATION

Exercice 1

Ecrire un programme permettant de résoudre dans $\Re$ une équation du premier degré de la forme **ax+b = 0** (traiter tous les cas possibles).

Exercice 2

Ecrire un programme permettant de résoudre dans $\Re$ une équation du second degré de la forme $\mathbf{ax^2+bx+c = 0}$ (traiter tous les cas possibles).

Exercice 3

Ecrire un programme qui lit un entier n compris entre 1 et 7 puis affiche le nom du jour correspondant si on suppose que le premier jour de la semaine est lundi.

Exercice 4

Ecrire un programme qui calcule le montant de la commission méritée par chacun des agents commerciaux et dont le taux varie selon le tableau suivant :

Montant total des ventes (TND)	Taux de la commission (%)
< 5000	1
[5000 .. 10000[	2
[10000 .. 20000[	3
>= 20000	4

Exercice 5

Ecrire un programme qui permet de vérifier si une année est bissextile ou non. Dans une année bissextile, le mois de février contient 29 jours ce qui fait un total de 366 jours.

Règle générale

- Les années divisibles par 100 sont bissextiles si elles sont divisibles également par 400.

A titre d'exemple, les années 2000 et 2400 sont bissextiles alors que l'année 2100 est une année ordinaire.

- Les autres années sont bissextiles si elles sont divisibles par 4.

A titre d'exemple, les années 2008, 2012 et 2016 sont bissextiles.

Exercice 6

Soit le programme suivant :

```
#include <iostream>
using namespace std;

int main()
{
	int n, p, i;
	cout<<"Entrer un entier :";
	cin>>n;
	p = 1;
	i = 1;
	while (i <= n)
	{
		p = p * 2;
		i++;
	}
	cout<<"Resultat = "<<p<<"\n";
	return 0;
}
```

1. Que vaudra le résultat fourni par le programme dans les cas suivants :

n	Résultat
0	?
2	?
3	?
5	?

2. Que fait ce programme ?
3. Ecrire une nouvelle version du programme en remplaçant la boucle *while* par une boucle *for*.

Exercice 7

Ecrire un programme qui lit un réel x et un entier positif n puis calcule et affiche x^n par multiplications successives.

$$x^n = x \times x \times \ldots \times x \quad \text{(n fois)}$$

Exemples

x	n	Résultat (x^n)
2	0	1
2	2	4
2	3	8
3.5	2	12.25
0	0	erreur

Exercice 8

Ecrire un programme qui calcule et affiche les 10 premiers termes de la suite de Fibonacci.

La suite de Fibonacci est définie par :

- $F_0 = 1$
- $F_1 = 1$
- $F_n = F_{n-2} + F_{n-1}$ pour $n > 1$

Exercice 9 : Nombres parfaits

Un nombre parfait est un nombre présentant la particularité d'être égal à la somme de tous ses diviseurs, excepté lui-même.

Le premier nombre parfait est 6 = 1 + 2 + 3.

Ecrire un programme qui affiche tous les nombres parfaits inférieurs à 1000.

Exercice 10

Ecrire un programme qui lit un entier n puis vérifie s'il s'agit d'un nombre premier ou non.

Un nombre premier possède exactement deux diviseurs (1 et lui-même).

Exercice 11

Ecrire un programme qui détermine tous les nombres premiers inférieurs à une valeur donnée.

Exercice 12 : Nombres cubiques

Parmi tous les entiers supérieurs à 1, seuls 4 peuvent être représentés par la somme des cubes de leurs chiffres.

Ainsi, par exemple : $153 = 1^3 + 5^3 + 3^3$ est un nombre cubique

Ecrire un programme qui permet de déterminer les 3 autres.

Note : les 4 nombres sont compris entre 150 et 410.

Exercice 13

Ecrire un programme qui calcule le PPCM (Plus Petit Commun Multiple) de 2 entiers A et B en utilisant la méthode suivante :

1. Permuter, si nécessaire, les données de façon à ranger dans A le plus grand des deux entiers
2. Chercher le plus petit multiple de A qui est au même temps un multiple de B.

Exemple : PPCM(6,8) = PPCM(8,6) = 24.

Exercice 14 : Nombres amis

Deux entiers a et b sont considérés comme amis si la somme des diviseurs de chacun d'eux est égale à l'autre.

A titre d'exemple, 220 et 284 sont deux nombres amis.

Ecrire un programme qui permet de trouver tous les nombres amis dans l'intervalle [1..2000].

Exercice 15

1. Ecrire un programme qui affiche la forme suivante à l'écran.

```
*
**
***
****
*****
******
*******
********
*********
**********
```

2. Modifier le programme afin d'obtenir la forme suivante :

```
**********
*********
********
*******
******
*****
****
***
**
*
```

Exercice 16

Ecrire un programme qui permet de générer les n premiers termes d'une suite arithmétique (a_n) de premier terme a_0 et de raison r.

On rappelle que le terme général d'une suite arithmétique est défini par la relation :

$$a_n = a_{n-1} + r = a_0 + n \times r$$

Exercice 17

Ecrire un programme qui permet de générer les n premiers termes d'une suite géométrique (g_n) de premier terme g_0 et de raison r.

On rappelle que le terme général d'une suite géométrique est défini par la relation :

$$g_n = g_{n-1} \times r = g_0 \times r^n$$

Les fonctions

1. La notion de fonction

On appelle *fonction* un sous-programme qui permet d'effectuer une tâche quelconque suite à un simple appel par n'importe quel programme ou n'importe quelle autre fonction. Les fonctions permettent de simplifier le code du programme principal et de réduire sa taille.

D'autre part, une fonction peut faire appel à elle-même, on parle alors de ***fonction récursive.***

2. Déclaration d'une fonction

Avant d'être utilisée, une fonction doit être déclarée car pour l'appeler dans le corps du programme, il faut que le compilateur la connaisse, c'est-à-dire qu'il connaisse son nom, ses arguments et les instructions qu'elle contient.

La déclaration d'une fonction se fait selon la syntaxe suivante :

```
type  Nom_Fonction(type1 arg1, type2 arg2, ...)
{
          corps de la fonction
}
```

- *type* représente le type de valeur que la fonction est sensée retourner (char, int, float, ...) ;
- Si la fonction ne renvoie aucune valeur, on la fait alors précéder du mot-clé *void* ;
- Si aucun type de données n'est précisé, le type *int* est pris par défaut ;
- Le nom de la fonction est un identificateur qui suit les mêmes règles que les noms de variables ;
- Les arguments sont facultatifs, mais les parenthèses doivent être présentes même s'il n'y a pas d'arguments.

Une fois cette étape franchie, la fonction ne s'exécutera pas tant qu'elle n'est pas appelée quelque part dans le programme.

3. Appel d'une fonction

Pour exécuter une fonction, il suffit de l'appeler en écrivant son nom (en respectant la casse) suivi d'une liste d'arguments entre parenthèses :

Nom_Fonction(arg1, arg2, ...);

Si la fonction a été définie sans arguments, il suffit de l'appeler en écrivant son nom suivi d'une parenthèse ouvrante et d'une parenthèse fermante :

Nom_Fonction();

Exemple

```
#include <iostream>
using namespace std;

int somme(int a, int b)    // déclaration de la fonction
{
      return a+b;
}

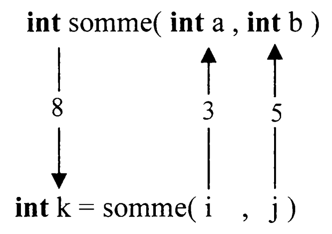
int main( )
{
      int i = 3,  j = 5;
      int k = somme(i,j);  // appel de la fonction
      cout<<"k = "<<k<<endl;
      return 0;
}
```

Output du programme

```
k = 8
```

- La fonction somme( ) retourne la somme de deux entiers fournis comme paramètres.

- Dans ce programme, le dialogue entre la fonction main( ) et la fonction somme( ) se fait selon le schéma suivant :



Exercice 1 : Ecrire une fonction ***sqr( )*** qui retourne le carré d'un nombre réel quelconque.

Solution

```
float sqr(float x)
{
      return x*x;
}
```

4. Renvoi d'une valeur par une fonction

La fonction peut renvoyer une valeur (et donc se terminer) grâce au mot-clé *return*. Lorsque l'instruction *return* est rencontrée, la fonction évalue l'expression qui la suit, puis renvoie la valeur au programme appelant (programme à partir duquel la fonction a été appelée).

Une fonction peut contenir plusieurs instructions *return*, ce sera toutefois la première instruction *return* rencontrée qui provoquera la fin de la fonction et le renvoi de la valeur qui la suit.

La syntaxe de l'instruction return est simple :

```
return expression;
```

Le type de valeur retourné doit correspondre à celui qui a été précisé dans la définition de la fonction.

Exemple

Ecrire une fonction ***min( )*** qui retourne le minimum de deux entiers a et b fournis en tant que paramètres.

Solution

```
int min(int a, int b)
{
        if (a <= b)
                return a;
        else
                return b;
}
```

Remarque

⚠ Le nombre et le type d'arguments dans la déclaration et dans l'appel d'une fonction doivent correspondre au risque, sinon, de générer une erreur lors de la compilation.

A titre d'exemple, tous les appels suivants de la fonction min( ) sont corrects :

- `m = min(3,5);`
- `m = min(min(3,2),4);`
- `m = min(min(3,6),min(2,5));`

alors que les appels suivants sont erronés puisqu'ils ne respectent pas cette règle :

- `m = min(3.5,5); // le premier paramètre sera convertit en entier`
- `m = min("pa","bx");`
- `m = min(3,2,5);`

Exercice

Ecrire une fonction ***max ( )*** qui retourne le maximum entre 3 nombres réels x, y et z.

Solution

```
float max(float x, float  y, float  z)
{
        float M = x;
        if (y > M)
                M = y;
        if (z > M)
                M = z;
        return M;
}
```

5. Variable locale et variable globale

Une **variable locale** (privée) est déclarée dans une fonction et ne peut être utilisée qu'à l'intérieur de celle-ci.

Une **variable globale** (publique) est déclarée au début du programme hors de toute fonction. Elle peut être utilisée n'importe où dans le programme.

Exemple

```
#include <iostream>
using namespace std;

float cel, fahr;            // variables globales

float Convert(float c)
{
      float f;              // variable locale
      f = c*9/5 + 32;
      return f;
}

int main()
{
      cel = 5;
      fahr = Convert(cel);
      cout<<"fahr = "<<fahr<<endl;
      return 0;
}
```

La fonction *convert( )* fait la conversion d'une température du degré Celsius vers le Fahrenheit.

Il est fortement recommandé d'utiliser autant que possible des variables locales pour rendre les fonctions plus autonomes et par conséquent utilisables dans n'importe quel programme et éviter les éventuels effets de bord résultant de la modification de la valeur des variables globales par les autres fonctions.

6. Passage des paramètres par valeur

Les paramètres utilisés lors de la définition d'une fonction s'appellent ***paramètres formels*** ou fictifs.

Les paramètres utilisés lors de l'appel d'une fonction s'appellent ***paramètres réels*** ou effectifs.

Par défaut, les paramètres d'une fonction sont initialisés par une copie des valeurs des paramètres réels. Ainsi, la modification de la valeur des paramètres formels dans le corps de la fonction ne change pas la valeur des paramètres réels.

Exemple

```
#include <iostream>
using namespace std;

void permut(int a,int b)
{
	int temp = a;
	a = b;
	b = temp;
}

int main ()
{
	int i = 5, j = 10;
	permut(i,j);
	cout<<"i = "<<i<<endl;
	cout<<"j = "<<j<<endl;
	return 0;
}
```

Output du programme

```
i = 5
j = 10
```

Le mode de passage de paramètres utilisé dans ce programme est le ***passage par valeur***.

La communication entre la fonction main( ) et la fonction permut( ) s'est déroulée selon le schéma suivant :

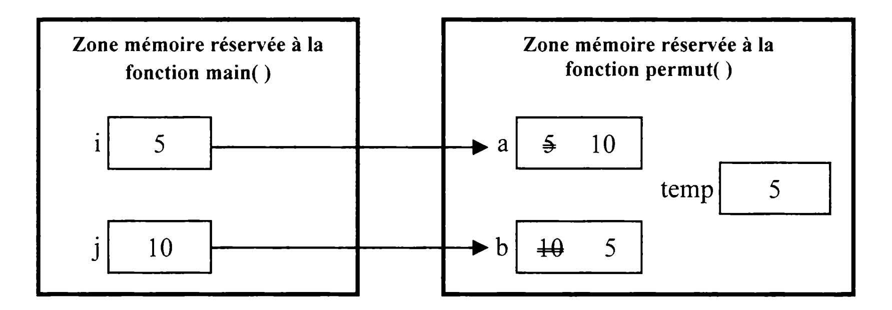


Figure 7 : Passage de paramètres par valeur

7. Passage des paramètres par référence

Pour modifier la valeur d'un paramètre réel dans une fonction, il faut passer ce paramètre par référence. Une *référence* sur une variable est un synonyme de cette variable, c'est-à-dire une autre manière de désigner le même emplacement de la mémoire.

On utilise le symbole **&** pour la déclaration d'une référence. Ainsi, dans la liste des paramètres de la définition d'une fonction, l'expression

$$\texttt{type \& } p_n$$

déclare le paramètre formel p_n comme étant une référence sur le $n^{ième}$ paramètre réel fourni lors de l'appel de la fonction.

Exemple

```
#include <iostream>

using namespace std;

void permut(int & a,int & b)
{
        int temp = a;
        a = b;
        b = temp;
}
```

```
int main ()
{
        int i = 5, j = 10;
        permut(i,j);
        cout<<"i = "<<i<<endl;
        cout<<"j = "<<j<<endl;
        return 0;
}
```

Output du programme

```
i = 10
j = 5
```

Dans ce dernier cas, la communication entre la fonction main( ) et la fonction permut( ) s'est déroulée selon le schéma suivant :

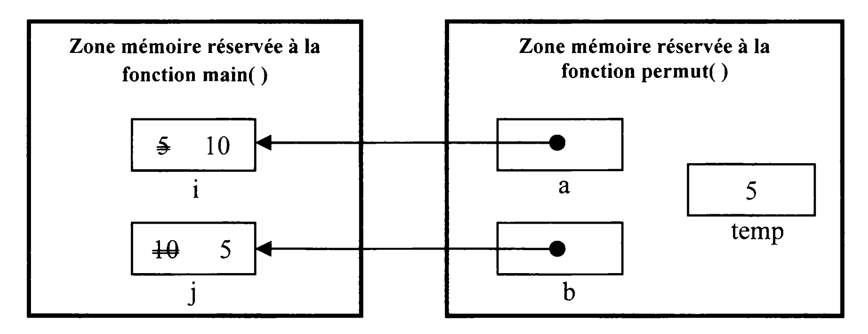


Figure 8 : Passage de paramètres par référence

Remarque : Le langage C++ offre un troisième mode de passage de paramètres : il s'agit du *passage par adresse* qui sera développé dans le chapitre réservé aux pointeurs (voir page 139).

8. Valeur par défaut des arguments

Pour simplifier les appels de fonctions comportant un paramètre qui varie peu, le langage C++ permet de déclarer des fonctions avec des paramètres qui ont des valeurs par défaut. La valeur par défaut d'un paramètre est la valeur que ce paramètre prend si aucune valeur ne lui est attribuée lors de l'appel de la fonction.

Exemple

```
#include <iostream>
using namespace std;

int test(int i = 0, int j = 2)
{
	return i/j;
}

int main ()
{
	int a = test(10,5);
	int b = test(8);
	int c = test();
	cout<<"a = "<<a<<endl;
	cout<<"b = "<<b<<endl;
	cout<<"c = "<<c<<endl;
	return 0;
}
```

Output du programme

```
a = 2
b = 4
c = 0
```

L'appel de la fonction test(8) est valide. Comme on ne précise pas le dernier paramètre, j est initialisé à 2. Le résultat obtenu est donc 4. De même, l'appel test( ) est valide, dans ce cas i vaut 0 et j vaut 2. En revanche, il est impossible d'appeler la fonction test( ) en ne précisant que la valeur de j.

Notons enfin que les paramètres par défaut doivent être les derniers de la liste, c'est-à-dire ceux le plus à droite. Ainsi l'expression

```
int test(int i=0, int j)
{...}
```

serait invalide, car si on ne passait pas deux paramètres, j ne serait pas initialisé.

9. Les fonctions inline

Le langage C++ dispose du mot clé *inline* qui permet de modifier la méthode d'implémentation des fonctions. Placé devant la déclaration d'une fonction, il propose au compilateur de ne pas instancier cette fonction. Cela signifie que l'on désire que le compilateur remplace l'appel de la fonction par le code correspondant.

Si la fonction est grosse ou si elle est appelée souvent, le programme devient volumineux puisque la fonction est réécrite à chaque fois qu'elle est appelée. En revanche, il devient nettement plus rapide puisque les mécanismes d'appel de fonctions, de passage des paramètres et de la valeur de retour sont ainsi évités. De plus, le compilateur peut effectuer des optimisations additionnelles qu'il n'aurait pas pu faire si la fonction n'était pas déclarée *inline*.

En pratique, on réservera cette technique pour les petites fonctions appelées dans du code devant être rapide (à l'intérieur des boucles par exemple), ou pour les fonctions permettant de lire des valeurs dans des variables.

Cependant, il faut connaître les restrictions des fonctions inline :

- elles ne peuvent pas être récursives ;
- elles ne sont pas instanciées, donc on ne peut pas créer un pointeur sur une fonction inline.

Exemple

```
inline int Max(int i, int j)
{
	if (i > j)
		return i;
	else
		return j;
}
```

Pour ce type de fonction, il est tout à fait justifié d'utiliser le mot clé *inline*.

10. Les fonctions récursives

La récursivité est une technique de programmation alternative à l'itération qui permet de trouver, lorsqu'elle est bien utilisée, des solutions très élégantes à un certain nombre de problèmes.

Une fonction est dite récursive si son corps contient un appel à elle-même :

```
Type f(paramètres)
{
        ...
        f(paramètres)
        ...
}
```

Utiliser la récursivité revient à définir :

- une solution pour un ensemble de cas de base (sans utiliser d'appels récursifs) ;
- une solution dans le cas général à travers une relation de récurrence avec des cas plus simples. Cette relation doit permettre d'arriver à un cas de base en un nombre fini d'étapes.

a. Etude d'un exemple : la fonction factorielle

Le factoriel d'un entier positif n se calcule en utilisant la formule suivante :

$n! = n \times (n-1) \times (n-2) \times \ldots \times 2$
$n! = n \times (n-1)!$

Sachant que $0! = 1$ et $1! = 1$.

Exemples

$3! = 6$
$4! = 4 \times 3! = 24$
$5! = 5 \times 4! = 120$

Le problème peut être alors défini par les deux relations suivantes :

- $n! = 1$ si $(n <= 1)$
- $n! = n \times (n-1)!$ si $(n > 1)$

Solution itérative	Solution récursive

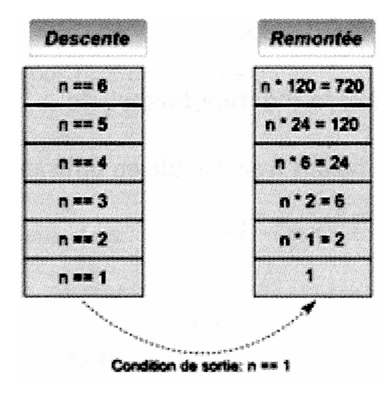
`int fact(int n)`
`{`
`   int f = 1;`
`   for (int i=n; i>=2; i--)`
`      f = f * i;`
`   return f;`
`}`
	`int fact(int n)`
	`{`
	`   if (n <= 1)`
	`      return 1;`
	`   else`
	`      return n*fact(n-1);`
	`}`

b. Mécanisme de fonctionnement de la récursivité

L'appel des fonctions récursives fonctionne à l'aide d'une pile d'exécution qui conserve les contextes d'appel :

1. à chaque appel de fonction, on empile le contexte : lieu de l'appel, variables locales, etc.
2. à chaque retour de fonction, on dépile le contexte, ce qui permet de revenir au point d'appel.

A titre d'exemple, le calcul de 6! par la fonction récursive *fact( )* définie précédemment se fait selon le schéma suivant :



Exercice 1 : Ecrire une fonction récursive `int PGCD(int a, int b)` qui calcule le PGCD de deux entiers positifs a et b en utilisant les relations suivantes :

- PGCD (a,b) = a si (a = b)
- PGCD (a,b) = PGCD(a-b, b) si (a > b)
- PGCD (a,b) = PGCD(a, b-a) si (b > a)

Solution

```
int PGCD(int a, int b)
{
	if (a==b)
		return a;
	else
		if (a > b)
			return PGCD(a-b, b);
		else
			return PGCD(a, b-a);
}
```

Exercice 2

Ecrire une fonction récursive `int Fibo(int n)` qui calcule le $n^{ième}$ terme de la suite de Fibonacci définie par :

- $F_0 = 1$
- $F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}$ si ($n > 1$)

Solution

```
int Fibo(int n)
{
	if (n <= 1)
		return 1;
	else
		return Fibo(n-1) + Fibo(n-2);
}
```

11. La surcharge de fonctions

Un des apports les plus intéressants du langage C++ par rapport au C, hormis l'ajout du concept objet, est la possibilité d'appeler plusieurs fonctions avec le même nom, pour peu que celles-ci aient des arguments différents (en type et/ou en nombre).

Ce principe est appelé *surcharge* de fonctions. Il permet de donner le même nom à des fonctions comportant des paramètres différents et simplifie donc l'écriture de fonctions sémantiquement similaires sur des paramètres de types différents.

En effet, une fonction est déterminée par ce que l'on appelle sa *signature*, c'est-à-dire :

- son nom
- ses paramètres.

Il est ainsi possible de définir une fonction réalisant la même opération sur des variables différentes en nombre et/ou en type.

Exemple

```
int somme(int p1, int p2)
{
        return (p1 + p2);
}
float somme(float p1, float p2)
{
        return (p1 + p2);
}
float somme(float p1, float p2, float p3)
{
        return (p1 + p2 + p3);
}
```

Exercice

Ecrire un programme qui contient deux fonctions qui portent le nom « surface » et dont :

- la première calcule la surface d'un rectangle à partir de la valeur de la longueur et de la largeur
- la seconde calcule la surface d'un cercle à partir de la valeur du rayon.

Solution

```
#include <iostream>
using namespace std;

float surface (float L, float l) // surface d'un rectangle
{
   return L*l;
}
```

```
float surface(float rayon)     // surface d'un cercle
{
   const float PI = 3.14;
   return PI * rayon * rayon;
}

int main( )
{
   float L, l;      // longueur et larguer du rectangle
   float r;         // rayon du cercle
   cout<<"Entrer la longueur du rectangle (en cm): ";
   cin>>L;
   cout<<"Entrer la largeur du rectangle (en cm): ";
   cin>>l;
   cout<<"Surface du rectangle = "<<surface(L,l)<<"cm2";
   cout<<endl;
   cout<<"Enter le rayon du cercle (en cm): ";
   cin>>r;
   cout<<"Surface du cercle = "<<surface(r)<<" cm2" ;
   cout<<endl;
   return 0;
}
```

Test du programme

```
Entrer la longueur du rectangle (en cm): 4
Entrer la largeur du rectangle (en cm): 3.5
Surface du rectangle = 14 cm2

Enter le rayon du cercle (en cm): 10
Surface du cercle = 314 cm2
```

12. Les macros

Outre la définition de constantes symboliques, le préprocesseur peut, lors du mécanisme de remplacement de texte, utiliser des paramètres fournis à l'identificateur à remplacer, ce qui permet de définir des macros de fonctions.

La syntaxe des macros est la suivante :

```
#define macro(paramètres) définition
```

Exemples

```
#define sum(a,b) a+b
#define max(a,b)((a >= b)? a : b)
```

Le mécanisme des macros permet de faire l'équivalent de fonctions générales, qui fonctionnent pour tous les types. Ainsi, la macro MAX renvoie le maximum de ses deux paramètres, qu'ils soient entiers, réels, caractères, etc.

Exercice (QCM)

1- Laquelle de ces affirmations est fausse ?

- O Une fonction n'est pas obligée de renvoyer une valeur
- O Une fonction peut renvoyer une valeur de n'importe quel type
- O Une fonction peut renvoyer plusieurs valeurs

2- Que sont les paramètres d'une fonction ?

- O Des indications sur le nom de la fonction
- O Des indications sur la valeur qu'elle doit renvoyer
- O Des variables qu'on lui envoie pour qu'elle puisse travailler

3- Que se passe-t-il après un return ?

- O La fonction s'arrête et renvoie le résultat indiqué
- O La fonction continue et renvoie le résultat indiqué
- O La fonction continue et ne renvoie pas de résultat

4- Dans quel cas l'instruction return n'est pas obligatoire ?

- O Quand la fonction ne prend aucun paramètre en entrée
- O Quand la fonction est de type void
- O Quand la fonction doit renvoyer 0

5- Quel est le problème de cette fonction qui est censée calculer le carré de la variable nombre ?

```
int carre(int nombre)
{
        int resultat = 0;
        resultat = nombre * nombre;
}
```

- O La fonction ne retourne aucune valeur
- O La fonction ne marche pas car on a oublié un point-virgule quelque part.

EXERCICES D'APPLICATION

Exercice 1

1. Ecrire une fonction qui porte le nom :

```
void replicate(char c, int n)
```

et qui permet d'afficher n fois le caractère c à l'écran.

A titre d'exemple, l'appel de fonction

```
replicate('A',3) ;
```

doit afficher 'AAA' à l'écran.

2. Utiliser la fonction replicate( ) pour afficher les formes suivantes sur l'écran :

```
*              ***********
**             **********
***            *********
****           ********
*****          *******
******         ******
*******        *****
********       ****
*********      ***
**********     **
***********    *
```

Exercice 2 : Ecrire une fonction qui retourne le PGCD de 2 entiers positifs a et b en utilisant l'algorithme d'Euclide qui consiste à répéter le traitement :

PGCD (a,b) = PGCD(b, a Mod b)

Jusqu'à obtenir la forme PGCD(x,0). Les PGCD est alors x.

Exemples

- PGCD(50,30) = PGCD(30,20) = PGCD(20,10) = PGCD(10,0) = 10
- PGCD(18,45) = PGCD(45,18) = PGCD(18,9) = PGCD(9,0) = 9

Exercice 3 : Ecrire une fonction qui retourne le PPCM (Plus Petit Commun Multiple) de 2 entiers a et b en utilisant la méthode suivante :

- Permuter, si nécessaire, les données de façon à ranger dans a le plus grand des 2 entiers ;
- Chercher le plus petit multiple de a qui est au même temps un multiple de b.

Exemple : PPCM(6,8) = PPCM(8,6) = 24.

Exercice 4 : Ecrire une fonction qui porte le nom :

```
bool Prime(int n)
```

et qui retourne la valeur *true* lorsque l'entier n est premier, sinon elle retourne la valseur *false*. On rappelle qu'un nombre premier possède uniquement deux diviseurs (un et lui-même).

Les tableaux statiques

1. La notion de tableau

Un *tableau* est une structure de données constituée d'un nombre fini d'éléments *de même type* et stockés de manière contiguë en mémoire (les uns à la suite des autres). Chaque élément du tableau occupe une case dont la taille est conditionnée par le type de ces éléments.

Voici donc une manière de représenter un tableau :

donnée	donnée	donnée	...	donnée	donnée	donnée

Les éléments du tableau peuvent être :

- des données de type simple : int, char, float, ... (la taille d'une case du tableau est alors égale au nombre d'octets sur lequel la donnée est codée) ;
- des pointeurs (objets contenant une adresse mémoire) ;
- des tableaux ;
- des structures.

Lorsque le tableau est composé de données de type simple, on parle de *tableau unidimensionnel* ou *vecteur*. Lorsque celui-ci contient lui-même d'autres tableaux on parle alors de *tableaux multidimensionnels* (appelés également *matrices* ou *tables*).

2. Les tableaux unidimensionnels

a. Déclaration

Un tableau unidimensionnel est une suite de « cases » de même taille contenant des éléments de type simple (int, float, char, ...).

En langage C++, la syntaxe de la définition d'un tableau unidimensionnel est la suivante :

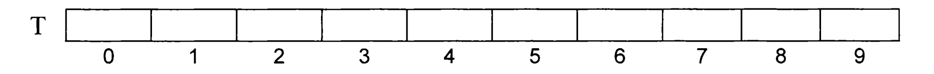
type Nom_Tableau[nombre d'éléments];

- *type* définit le type d'élément que contient le tableau, c'est-à-dire qu'il définit la taille d'une case du tableau en mémoire ;
- *Nom_Tableau* est le nom que l'on décide de donner au tableau, il suit les mêmes règles qu'un nom de variable ;
- *nombre d'éléments* est un nombre entier qui détermine le nombre de cases que le tableau doit contenir.

Voici par exemple la définition d'un tableau qui doit contenir 10 éléments de type entier :

```
int T[10];
```

Les éléments de ce tableau seront numérotés de 0 à 9 et peuvent être représentés de la façon suivante :



b. Accès aux éléments

Pour accéder à un élément du tableau, le nom que l'on a donné à celui-ci ne suffit pas car le tableau contient plusieurs éléments. Ainsi, on définit un nombre appelé **indice** ou **index** qui, combiné avec le nom du tableau, permet de décrire exactement chaque élément.

Pour accéder à un élément du tableau, il suffit donc de donner le nom du tableau, suivi de l'indice de l'élément entre crochets :

Nom_Tableau[indice]

Remarques

1. L'indice du premier élément du tableau est toujours 0.
2. L'indice du dernier élément du tableau est égal au nombre d'éléments – 1. Ainsi, on accèdera au 5ème élément du tableau T en écrivant :

```
T[4]
```

c. Manipulation des éléments

Un élément du tableau (repéré par le nom du tableau et son indice) peut être manipulé exactement comme une variable du même type, on peut donc lui faire appliquer toutes les opérations valables sur ces variables.

Exemple : Soit T un tableau de 10 entiers défini comme suit :

```
int T[10];
```

- Pour affecter la valeur 6 au huitième élément du tableau, on écrira :

```
T[7] = 6;
```

- Pour affecter au 10ème élément le résultat de l'addition du 1er et du 2ème élément, on écrira :

```
T[9] = T[0] + T[1];
```

d. Initialisation des éléments

Lorsque l'on définit un tableau, les valeurs des éléments qu'il contient ne sont pas définies, il faut donc les initialiser, c'est-à-dire leur affecter une valeur.

Une méthode rustique consiste à affecter des valeurs aux éléments un par un :

```
T[0] = T[1] = T[2] = 0;
```

L'intérêt de l'utilisation d'un tableau est alors bien maigre.

Une manière plus élégante consiste à utiliser le fait que pour passer d'un élément du tableau à l'élément suivant il suffit d'incrémenter son indice. Il est donc possible d'utiliser une boucle qui va permettre d'initialiser successivement chacun des éléments grâce à un compteur qui servira d'indice :

```
int T[10];
int i;
for (i = 0; i < 10; i++)
{
        T[i] = 0;
}
```

Cette méthode, aussi utile soit elle, n'a d'intérêt que lorsque les éléments du tableau doivent être initialisés à une valeur unique ou une valeur logique (proportionnelle à l'indice par exemple). Pour initialiser un tableau avec des valeurs spécifiques, il suffit de placer ces valeurs entre des accolades, séparées par des virgules, comme dans l'exemple suivant :

```
int T[10] = {1, 2, 6, 5, 2, 1, 9, 8, 1, 5};
```

Remarques

1. Le nombre de valeurs entre accolades ne doit pas être supérieur au nombre d'éléments du tableau.
2. Les valeurs entre accolades doivent être des constantes (l'utilisation de variables provoquera une erreur du compilateur).
3. Si le nombre de valeurs entre accolades est inférieur au nombre d'éléments du tableau, les derniers éléments seront initialisés à 0.
4. Il doit y avoir au moins une valeur entre accolades. Ainsi, l'instruction suivante permet d'initialiser tous les éléments du tableau à zéro :

```
int T[10] = {0};
```

Exemple

Ecrire un programme qui remplit un tableau de 5 entiers entrés à partir du clavier puis affiche ces éléments dans l'ordre inverse de leur entrée.

Solution

```
#include <iostream>
using namespace std;

int main()
{
      int T[5];
      int i;

      // Remplissage du tableau
      for (i = 0; i < 5; i++)
      {
            cout<<"Entrer un entier : ";
            cin>>T[i];
      }
```

```
        // Affichage des éléments dans l'ordre inverse
        for (i = 4; i >= 0; i--)
        {
                cout<<T[i]<<"\t";
        }
        cout<<endl;
        return 0;
}
```

Test du programme

```
Entrer un entier : 10
Entrer un entier : 15
Entrer un entier : 20
Entrer un entier : 30
Entrer un entier : 40

40      30      20      15      10
```

Exercice

Ecrire un programme qui permet de déterminer à partir de 20 notes fournies en entrée, la moyenne des notes et le nombre d'étudiants qui ont obtenu une note supérieure à cette moyenne.

Solution

```
#include <iostream>

using namespace std;

int main()
{
        float Tnote[20];
        float Somme, Moyenne;
        int nb, i;

        // Remplissage du tableau
        Somme = 0;
        for (i = 0; i < 20; i++)
        {
                cout<<"Entrer une note : ";
                cin>>Tnote[i];
                Somme = Somme + Tnote[i];
        }

        // Calcul de la moyenne
        Moyenne = Somme / 20;
        cout<<"Moyenne de la classe = "<<Moyenne<<endl;
```

```
        /* Comptage du nombre d'étudiants qui ont obtenu une
        note supérieure à la moyenne */
        nb = 0;
        for (i = 0; i < 20; i++)
        {
                if (Tnote[i] > Moyenne)
                        nb++;
        }
        cout<<nb<<" etudiants ont obtenu une note
        superieure a la moyenne";
        cout<<endl;
        return 0;
}
```

Test du programme

```
Entrer une note : 10
Entrer une note : 12
Entrer une note : 12.5
Entrer une note : 16
Entrer une note : 8
Entrer une note : 12
Entrer une note : 14
Entrer une note : 9
Entrer une note : 7.5
Entrer une note : 10
Entrer une note : 11
Entrer une note : 13.5
Entrer une note : 12
Entrer une note : 16
Entrer une note : 7
Entrer une note : 15
Entrer une note : 16
Entrer une note : 13
Entrer une note : 8
Entrer une note : 10

Moyenne de la classe = 11.625

11 etudiants ont obtenu une note superieure a la moyenne
```

Exercice

Soit T un tableau contenant n éléments de type entier. Ecrire une fonction dont l'en-tête est de la forme :

```
int MinTab(int T[], int n)
```

et qui retourne le plus petit élément de ce tableau.

Solution

On suppose initialement que le premier élément du tableau est le minimum puis on le compare à tous les autres éléments. Chaque fois qu'on trouve un élément qui lui est inférieur, ce dernier devient le minimum.

```
int MinTab(int T[], int n)
{
        int min = T[0];
        for (int i = 1; i < n; i++)
        {
                if (T[i] < min)
                        min = T[i];
        }
        return min;
}
```

Exercice

Ecrire une fonction qui remplit un tableau Fib par les n premiers termes de la suite de Fibonacci.

La suite de Fibonacci est définie par :

- $F_0 = 1$
- $F_1 = 1$
- $F_n = F_{n-2} + F_{n-1}$ pour $n > 1$

Solution

```
void remplir(int Fib[], int n)
{
        Fib[0] = 1;
        Fib[1] = 1;
        for (int i = 2; i < n; i++)
                Fib[i] = Fib[i-2] + Fib[i-1];
}
```

3. Méthodes de recherche

a. Recherche séquentielle

Soit T un tableau contenant n éléments de type entier.

On veut écrire une fonction dont l'en-tête est de la forme :

```
int recherche(int x, int T[], int n)
```

Cette fonction doit retourner :

- l'indice de la première occurrence de x dans T si $x \in T$
- la valeur (-1) si $x \notin T$.

Principe

La recherche séquentielle consiste à examiner à tour de rôle chaque élément jusqu'à la réussite de la recherche ou l'épuisement des éléments du tableau.

Solution

```
int recherche(int x, int T[], int n)
{
      for (int i = 0; i < n; i++)
      {
            if (T[i] == x)       // recherche réussi
                  return i;
      }
      return -1;                 // valeur introuvable
}
```

Exercice

Ecrire une fonction dont l'en-tête est de la forme :

```
int frequence(int x, int T[], int n)
```

et qui permet de compter combien de fois l'entier x existe dans le tableau T de taille n.

Solution

```
int frequence(int x, int T[], int n)
{
        int nb = 0;
        for (int i=0; i < n; i++)
        {
             if (T[i] == x)
                 nb++;
        }
        return nb;
}
```

b. Recherche dichotomique

Soit T un tableau contenant n éléments de type entier triés dans le sens croissant :

T	1	3	5	5	8	9	11	14	15	20
	0	1	2	3	4	5	6	7	8	9

On veut écrire une fonction dont l'en-tête est de la forme :

```
int RechDicho(int x, int T[], int n)
```

Cette fonction doit retourner :

- l'indice de la première occurrence de x trouvée dans T si $x \in T$
- la valeur (-1) si $x \notin T$

Principe

Le but de la recherche dichotomique est de diviser l'intervalle de recherche par 2 à chaque itération. Pour cela, on procède de la façon suivante :

Soient *premier* et *dernier* les extrémités gauche et droite de l'intervalle dans lequel on cherche la valeur x; on calcule M, l'indice de l'élément médian :

M = (premier + dernier) / 2

Il y a 3 cas possibles :

- (x = T[M]) : l'élément de valeur x est trouvé, la recherche est terminée.

- (x < T[M]) : l'élément x, s'il existe, se trouve dans l'intervalle [premier..M-1].
- (x > T[M]) : l'élément x, s'il existe, se trouve dans l'intervalle [M+1..dernier].

La recherche dichotomique consiste à itérer ce processus jusqu'à ce que l'on trouve x ou l'intervalle de recherche devient vide.

Solution

```
int RechDicho(int x, int T[], int n)
{
        int premier = 0;
        int dernier = n-1;
        int M;
        do
        {
                M = (premier + dernier) / 2;
                if (T[M] == x)
                        return M;
                if (x < T[M])
                        dernier = M - 1;
                else
                        premier = M + 1;
        } while (premier <= dernier);
        return -1;
}
```

4. Méthodes de tri

a. Tri à bulle

Soit T un tableau contenant n éléments de type entier :

T	6	4	3	5	2
	0	1	2	3	4

On veut écrire une fonction dont l'en-tête est de la forme :

```
void TriBulle(int T[], int n)
```

et qui permet de trier les éléments du tableau dans le sens croissant en utilisant la méthode de tri à bulle.

Principe

La méthode de tri à bulles nécessite deux étapes :

1. Comparer les éléments du tableau par paires adjacentes ; chaque fois qu'on trouve deux éléments non triés dans le sens désiré, on les permute.

2. Si au moins une permutation a été faite durant le parcours précédent, on revient à l'étape 1 et on recommence un nouveau parcours ; sinon on s'arrête (le tableau est trié).

A titre d'exemple, la fonction ci-dessous permet de trier un tableau T de n éléments dans le ses croissant :

```
void Tri_Bulle(int T[], int n)
{
      int  i,  NbPerm,  temp;
      do
      {
            NbPerm = 0;
            for (i = 0; i < (n-1); i++)
            {
                  if (T[i] > T[i+1])
                  {
                        temp = T[i];
                        T[i] = T[i+1];
                        T[i+1] = temp;
                        NbPerm++;
                  }
            }
      } while (NbPerm > 0);
}
```

Trace d'exécution

Tableau initial	6	4	3	5	2
Après la 1ère itération	4	3	5	2	6
Après la 2ème itération	3	4	2	5	6
Après la 3ème itération	3	2	4	5	6
Après la 4ème itération	2	3	4	5	6

b. Tri par sélection (par minimum)

Principe

C'est l'une des méthodes de tri les plus simples, elle consiste à :

- chercher l'indice du plus petit élément du tableau T[0..n-1] et permuter l'élément correspondant avec l'élément T[0] ;
- chercher l'indice du plus petit élément du tableau T[1..n-1] et permuter l'élément correspondant avec l'élément T[1] ;
- ...
- chercher l'indice du plus petit élément du tableau T[n-2..n-1] et permuter l'élément correspondant avec l'élément T[n-2].

```
void Tri_Selection(int T[], int n)
{
      int  i,  j,  IndiceMin, temp;
      for (i = 0; i < (n-1); i++)
      {
            IndiceMin = i;
            for (j = i+1; j < n; j++)
            {
                  if (T[j] < T[IndiceMin])
                        IndiceMin = j;
            }
            temp = T[i];
            T[i] = T[IndiceMin];
            T[IndiceMin] = temp;
      }
}
```

Trace d'exécution

Tableau initial	6	4	3	5	2
Après la 1ère itération	2	4	3	5	6
Après la 2ème itération	2	3	4	5	6
Après la 3ème itération	2	3	4	5	6
Après la 4ème itération	2	3	4	5	6

c. Tri par insertion

Principe

Cette méthode consiste à prendre les éléments de la liste un par un et insérer chacun dans sa bonne place de telle sorte que les éléments traités forment une sous-liste triée.

Pour ce faire, on procède de la façon suivante :

- comparer et permuter si nécessaire T[0] et T[1] de façon à placer le plus petit dans la case d'indice 0 et former une sous-liste triée T[0..1] ;
- comparer et permuter si nécessaire l'élément T[2] avec ceux qui le précèdent dans l'ordre (T[1] puis T[0]) afin de former une sous-liste triée T[0..2] ;
- ...
- comparer et permuter si nécessaire l'élément T[n-1] avec ceux qui le précèdent dans l'ordre (T[n-2], T[n-3], ...) afin d'obtenir un tableau trié.

```
void Tri_Insertion(int T[], int n)
{
      int  i,  j, temp;
      for (i = 1; i < n; i++)
      {
            temp = T[i];
            j = i - 1;
            while ((j >= 0) && (T[j] > temp))
            {
                  T[j+1] = T[j];
                  j = j - 1;
            }
            T[j+1] = temp;
      }
}
```

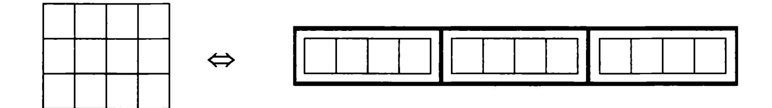
Trace d'exécution

Tableau initial	6	4	3	5	2
Après la 1ère itération	4	6	3	5	2
Après la 2ème itération	3	4	6	5	2
Après la 3ème itération	3	4	5	6	2
Après la 4ème itération	2	3	4	5	6

5. Les tableaux multidimensionnels

Les tableaux multidimensionnels sont des tableaux qui contiennent des tableaux.

Par exemple, le tableau bidimensionnel (3 lignes, 4 colonnes) suivant est en fait un tableau comportant 3 éléments, chacun d'entre eux étant un tableau de 4 éléments :



a. Déclaration

On déclare un tableau multidimensionnel de la manière suivante :

type Nom_Tableau [taille1][taille2][taille3] ... ;

- Chaque élément entre crochets désigne le nombre d'éléments dans une dimension ;
- Le nombre de dimensions n'est pas limité.

A titre d'exemple, un tableau d'entiers à deux dimensions (3 lignes, 4 colonnes) se définira avec la syntaxe suivante :

```
int T[3][4];
```

On peut représenter un tel tableau de la manière suivante :

T[0][0]	T[0][1]	T[0][2]	T[0][3]
T[1][0]	T[1][1]	T[1][2]	T[1][3]
T[2][0]	T[2][1]	T[2][2]	T[2][3]

b. Initialisation des éléments

L'initialisation d'un tableau multidimensionnel se fait à peu près de la même façon que pour les tableaux unidimensionnels. Il y a donc plusieurs façons d'initialiser un tableau multidimensionnel :

- ***Initialisation individuelle de chaque élément***

```
T[0][0] = a1;
T[0][1] = a2;
...
```

- ***Initialisation grâce à des boucles***

Il faut utiliser des boucles imbriquées correspondant chacune à un indice d'une dimension. Par exemple, le code suivant permet d'initialiser les éléments du tableau T[3][4] à 0 :

```
int T[3][4];
int i, j;
for (i=0; i<3; i++)         // parcours des lignes
{
        for (j=0; j<4; j++) // parcours des colonnes
        {
                T[i][j] = 0;
        }
}
```

- ***Initialisation à la définition***

```
type Nom_Tableau[Taille1][Taille2]... = {a1, a2, ...};
```

Les valeurs sont attribuées aux éléments successifs en incrémentant d'abord les indices de droite, c'est-à-dire pour un tableau à 2 dimensions : [0][0], [0][1], [0][2] ... puis [1][0], [1][1], etc.

Exemple

```
int T[4][3] = {0,1,2, 3,4,5, 6,7,8, 9,10,11};
```

T	0	1	2
0	0	1	2
1	3	4	5
2	6	7	8
3	9	10	11

Exercice 1

Comment sera le contenu du tableau M après l'exécution du code suivant :

```
int M[10][10];
int i, j;
for (i = 0; i < 10; i++)
    for (j = 0; j < 10; j++)
        M[i][j] = i * j;
```

Solution

M	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	10	12	14	16	18
3	0	3	6	9	12	15	18	21	24	27
4	0	4	8	12	16	20	24	28	32	36
5	0	5	10	15	20	25	30	35	40	45
6	0	6	12	18	24	30	36	42	48	54
7	0	7	14	21	28	35	42	49	56	63
8	0	8	16	24	35	40	48	56	64	72
9	0	9	18	27	36	45	54	63	72	81

Exercice 2 : Transposition d'une matrice carrée

Une matrice carrée est un tableau à n lignes et n colonnes.

L'opération de transposition consiste à inverser les lignes et les colonnes en effectuant une symétrie par rapport à la diagonale principale de la matrice.

<u>Exemple</u>

M

1	2	3
4	5	6
7	8	9

devient

1	4	7
2	5	8
3	6	9

Ecrire une fonction qui fait la transposition d'une matrice carrée quelconque.

Solution

```
void transpose(int M[n][n])
{
	int i, j, temp;
	for (i = 0; i < n; i++)
	{
		for (j = (i+1); j < n; j++)
		{
			temp = M[i][j];
			M[i][j] = M[j][i];
			M[j][i] = temp;
		}
	}
}
```

Exercice 3 : Somme de deux matrices

Soient M1 et M2 deux matrices à n lignes et m colonnes.

Ecrire une fonction SomMat( ) qui calcule les éléments de la matrice M3=M1+M2.

Exemple

M1

1	2	3
4	5	6

et M2

2	5	3
3	0	1

donnent M3

3	7	6
7	5	7

Solution

```
void SomMat(int M1[n][m], int M2[n][m], int M3[n][m])
{
	int i, j;
	for (i = 0; i < n; i++)
	{
		for (j = 0; j < m; j++)
		{
			M3[i][j] = M1[i][j]+ M2[i][j];
		}
		cout<<endl;
	}
}
```

Exercice 4 : Produit de deux matrices

Soient M1 une matrice ayant n lignes et m colonnes
M2 une matrice ayant m lignes et p colonnes

Ecrire une fonction ProdMat( ) qui calcule les éléments de la matrice M3=M1×M2.

Solution

Notons d'abord que le nombre de colonnes de M1 doit être égal au nombre de lignes de M2.

Le produit M3 = M1 × M2 est défini comme une matrice ayant n lignes et p colonnes et dont les éléments sont calculés par la formule :

$$M3_{i,j} = M1_{i,1}M2_{1,j} + M1_{i,2}M2_{2,j} + \dots + M1_{i,m}M2_{m,j}$$

$$M3_{i,j} = \sum_{k=1}^{m} M1_{i,k}M2_{k,j}$$

où $M1_{i,k}$, $M2_{k,j}$ et $M3_{i,j}$ sont respectivement les éléments des matrices M1, M2 et M3.

Exemple

M1

1	2	3
4	0	5

et M2

2	1
3	0
1	4

donnent M3

11	13
13	24

Solution

```
void ProdMat(int M1[n][m], int M2[m][p], int M3[n][p])
{
   int i, j, k;
   for (i = 0; i < n; i++)
   {
      for (j = 0; j < p; j++)
      {
         M3[i][j] = 0;
         for (k = 0; k < m; k++)
         {
            M3[i][j] = M3[i][j] + M1[i][k] * M2[k][j];
         }
      }
   }
}
```

Exercice (QCM)

1- Laquelle de ces lignes crée un tableau de 10 éléments de type double ?

- ○ `double* tableau[10];`
- ○ `double tableau{10};`
- ○ `double tableau[10];`
- ○ `double tableau[9];`

2- A quel indice commence les éléments d'un tableau ?

- ○ A l'indice 0
- ○ A l'indice 1
- ○ A l'indice -1

3- Lequel de ces prototypes de fonction ne permet pas de faire passer un tableau en tant que paramètre?

- ○ `void fonction(int tableau[], int taille);`
- ○ `void fonction(int tableau, int taille);`

4- Si je crée un tableau de 20 éléments de type char appelé maChaine situé à l'adresse 45782015, quelle est l'adresse de maChaine[3] ?

- ○ 45782015
- ○ 45782016
- ○ 45782017
- ○ 45782018
- ○ 45782019

5- Quelle est l'autre façon d'initialiser le tableau Tab avec ces valeurs ?

```
int Tab[4];
Tab[0] = 10;
Tab[1] = 23;
Tab[2] = 505;
Tab[3] = 8;
```

- O `int Tab[4] = 10, 23, 505, 8;`
- O `int Tab[4] = [10, 23, 505, 8];`
- O `int Tab[4] = (10, 23, 505, 8);`
- O `int Tab[4] = {10, 23, 505, 8};`
- O `int Tab[ ] = {10, 23, 505, 8};`

EXERCICES D'APPLICATION

Exercice 1

Ecrire une fonction permettant d'éclater un tableau T d'entiers en deux tableaux :

- TP qui contiendra les éléments positifs de T
- TN qui contiendra les éléments négatifs de T.

Exercice 2

Un instituteur cherche à vérifier si ses élèves ont appris à réciter l'alphabet lettre par lettre dans l'ordre. Pour cela, Il vous demande de lui développer un programme permettant d'évaluer chaque élève de la façon suivante :

1- Le programme demande à l'élève de remplir un tableau nommé réponse par les lettres de l'alphabet dans l'ordre

2- Le programme examine ensuite le tableau élément par élément :

- Si la lettre est dans sa place, il l'accepte ;
- Sinon, il la remplace par la lettre adéquate et incrémente le nombre de fautes.

3- Le programme affiche enfin le nombre total de fautes.

Exercice 3 : Produit scalaire de 2 vecteurs

Ecrire une fonction dont l'en-tête est de la forme :

```
int ProdScal(int U[], int V[], int n)
```

qui calcule le produit scalaire de deux vecteurs U et V représentés par deux tableaux.

Le produit scalaire de deux vecteurs :

$$U = (x_0, x_1, \ldots, x_{n-1}) \quad \text{et} \quad V = (y_0, y_1, \ldots, y_{n-1})$$

est défini par :

$$U.V = x_0y_0 + x_1y_1 + \ldots + x_{n-1}y_{n-1} = \sum_{i=0}^{n-1} x_iy_i$$

Exercice 4 : Norme d'un vecteur

Ecrire une fonction dont l'en-tête est de la forme :

```
int NormVect(int U[], int n)
```

qui fait appel à la fonction *ProdScal( )* de l'exercice précédent pour calculer la norme d'un vecteur U quelconque.

La norme d'un vecteur $U = (x_0, x_1, \ldots, x_{n-1})$ est définie par :

$$\|U\| = \sqrt{x_0^2 + x_1^2 + \ldots + x_{n-1}^2}$$

Exercice 5 : Compression d'un vecteur

Soit un vecteur U de composantes x_0, x_1, …, x_{n-1} non nulles et un vecteur L de même longueur dont les composantes sont 0 ou 1.

Ecrire une fonction qui fait la compression de U par L. Le résultat est un vecteur V dont les composantes sont, dans l'ordre, celles de U pour lesquelles la composante de L vaut 1.

Exemple

U	1	2	3	4	5	6	7
L	0	1	1	0	1	0	1
V	2	3	5	7	0	0	0

Exercice 6 : Recherche de la plus grande monotonie dans un tableau

Soit un tableau T de n éléments, déterminer la longueur de la première plus longue séquence de nombres rangés par ordre croissant et le rang de son premier élément.

Exercice 7 : Fusion de deux tableaux triés

Ecrire une fonction qui permet de fusionner deux tableaux triés A et B contenant respectivement n et m éléments. Le résultat est un tableau trié C à (n+m) éléments.

Exemple

A	1	20	41

B	19	23	27	54	91

C	1	19	20	23	27	41	54	91

Exercice 8

Etant donné un tableau A de n nombres réels triés par ordre croissant et R un réel quelconque. Ecrire une fonction qui permet d'insérer le nombre R dans sa bonne position. Le résultat sera un deuxième tableau B qui contient (n+1) éléments également triés par ordre croissant.

Exemple

A	1	2	5	8	10	25

R = 6

B	1	2	5	6	8	10	25

Exercice 9 : Triangle de Pascal

Créer un tableau à deux dimensions qui contiendra les n premières lignes du triangle de Pascal.

Les éléments du triangle de pascal sont reliés par la formule :

T[L,C] = T[L-1,C-1] + T[L-1,C]   (pour L > 0 et C > 0)

A titre d'exemple, pour n = 6, le triangle de pascal est :

1	0	0	0	0	0
1	1	0	0	0	0
1	2	1	0	0	0
1	3	3	1	0	0
1	4	6	4	1	0
1	5	10	10	5	1

Exercice 10 : La tour sur un échiquier

Sur un échiquier, c'est-à-dire un tableau de 8 lignes et 8 colonnes, à partir d'une case quelconque, marquer toutes les cases susceptibles d'être atteintes en un coup par une tour. Au terme de l'exécution, les cases atteintes contiendront la valeur 1, les autres la valeur 0.

La tour peut se déplacer sur la ligne et la colonne issues de la case où elle est située.

Exemple : Si la tour se trouve à la case (L=1 , C=2), le tableau final aura l'allure suivante :

0	0	1	0	0	0	0	0
1	1	1	1	1	1	1	1
0	0	1	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	0	0	0	0	0

Exercice 11 : Le fou sur un échiquier

Sur un échiquier, c'est-à-dire un tableau de 8 lignes et 8 colonnes, à partir d'une case quelconque, marquer toutes les cases susceptibles d'être atteintes en un coup par un fou. Au terme de l'exécution, les cases susceptibles d'être atteintes contiendront la valeur 1, les autres la valeur 0.

Le fou peut se déplacer sur les diagonales issues de la case où il est situé.

<u>Exemple</u>

Si le fou se trouve à la case (L=3 , C=2), le tableau final aura l'allure suivante :

0	0	0	0	0	1	0	0
1	0	0	0	1	0	0	0
0	1	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	1	0	1	0	0	0	0
1	0	0	0	1	0	0	0
0	0	0	0	0	1	0	0
0	0	0	0	0	0	1	0

<u>Remarque</u>

Pour toutes les cases qui peuvent être atteintes par le fou, l'une des 2 conditions suivantes est vérifiée :

- L – C = 3 - 2 = 1
- L + C = 3 + 2 = 5

Les tableaux dynamiques

Introduction

En C++, il est possible de créer des tableaux **statiques** (c'est-à-dire dont la taille ne varie pas) en utilisant les crochets [ ]. Si l'on veut créer des tableaux dont la taille **varie** au cours du programme ou si la taille d'un tableau n'est pas connue lors de la compilation, des problèmes apparaissent. Pour créer des tableaux dynamiques, il faut allouer soi-même la mémoire via l'opérateur *new* et redimensionner « à la main » le tableau à chaque fois que l'on ajoute un élément ou qu'on en supprime. C'est donc très peu pratique et l'on fait souvent des erreurs.

Pour éviter tous ces problèmes, la bibliothèque standard du langage C++ fournit une classe spéciale pour gérer les tableaux dynamiques. Il s'agit de la classe **std::vector**.

1. Bibliothèque et espace de nom

Pour utiliser les tableaux dynamiques, il faut inclure le fichier d'en-tête *vector* qui les définit en écrivant :

```
#include <vector>
```

Le template <vector> est un conteneur standard, il se trouve par conséquent dans l'espace de nom *std*. Il y a alors deux possibilités :

1. ajouter `std::` devant chaque fonction
2. utiliser la directive `using namespace std;`

On utilisera la deuxième méthode dans ce manuel afin de rendre le code le plus clair possible.

2. Création d'un tableau dynamique

Pour créer un tableau typé, on utilise la syntaxe suivante :

vector<type> NomTab(n);

avec :

- *type* : le type des éléments du tableau (int, float, …) ;
- *NomTab* : le nom du tableau ;
- *n* : la taille du tableau.

Dans ce cas, les éléments du tableau seront automatiquement initialisés à 0. Pour les initialiser à une valeur v quelconque, on utilise la syntaxe suivante :

vector<type> NomTab(n,v);

Exemples

```
vector<int> T1(5); // Crée un tableau de 5 entiers initialisés à 0

vector<int> T2(10,2) ; // Crée un tableau de 10 entiers initialisés à 2

vector<double> T3 ; // Crée un tableau vide pouvant contenir des réels
```

3. Accès aux éléments du tableau

Pour accéder aux éléments du tableau, il suffit d'utiliser les crochets [ ] comme dans le cas des tableaux statiques. Le premier élément possède l'indice **0**. Ce qui donne par exemple :

```
Tab[5] = 7; // Met la valeur 7 dans la 6ème case du tableau Tab.

cout<<Tab[0]; // Affiche la valeur de la 1ère case du tableau Tab.
```

Il existe également deux fonctions membres permettant d'accéder à des cases particulières du tableau. La première, `front()`, permet d'accéder au premier

élément du tableau. La deuxième, `back()`, permet d'accéder au dernier élément sans avoir besoin de connaître la taille du tableau.

Exemple

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
        vector<double> T(10,0);
        T.front()= 2.718;
        T.back() = 1;
        cout <<"T[0] = "<<T[0]<< endl;
        cout <<"T[9] = "<<T[9]<< endl;
        return 0;
}
```

Output du programme

```
T[0] = 2.718
T[9] = 1
```

4. Quelques fonctions membres

Le tableau 13 présente un résumé des principales fonctions membres (méthodes) de la classe vector qui sont utilisées pour la manipulation des tableaux dynamiques.

Tableau 13 : Fonctions de manipulation de tableaux dynamiques

Fonction membre	Description
T.size()	renvoie la taille du tableau T
T.empty()	renvoie la valeur true (1) si le tableau est vide et false (0) dans le cas contraire
T.push_back(x)	ajoute une case à la fin du tableau et la remplit avec la valeur passée en argument
T.pop_back()	supprime la dernière case du tableau
T.clear()	vide entièrement le tableau
T.resize(nb)	permet de redimensionner le tableau T
T.assign(n,v)	vide entièrement le tableau et remplace son contenu par n éléments dont la valeur est v.

Exemple

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
	vector<int> v(5);
	v[1]= 10;
	v[3]= 30;
	v.resize(6);
	v[5]= 50;
	v.push_back(60);
	for (int i=0; i<v.size(); i++)
	{
		cout <<"v["<<i<<"] = "<<v[i]<< endl;
	}
	return 0;
}
```

Output du programme

```
v[0] = 0
v[1] = 10
v[2] = 0
v[3] = 30
v[4] = 0
v[5] = 50
v[6] = 60
```

5. Passage d'un tableau à une fonction

Les tableaux de la classe *std::vector* étant des objets comme les autres, il est possible de les passer en argument à une fonction. Le prototype d'une telle fonction serait par exemple :

```
#include <vector>
using namespace std;

void f(vector<int> tableau);
```

Le programmeur n'a pas besoin de passer la taille comme deuxième argument, comme dans le cas des tableaux statiques, puisqu'on peut accéder à cette information via la fonction membre `size()`.

Cette manière de faire est cependant une mauvaise idée. En effet, en procédant de la sorte, Le tableau sera recopié entièrement lors du passage à la fonction. Il y a donc une perte de temps (qui peut être importante si le tableau est grand) et surtout une duplication de l'information inutile. La bonne solution consiste à utiliser une référence ou mieux encore, une référence constante si on ne compte pas modifier le tableau dans la fonction.

```
#include <vector>
using namespace std;

void f(vector<int>& tableau); // Bien

void g(const vector<int>& tableau); //Encore mieux
```

Passer des arguments par référence constante est une bonne habitude à prendre en C++.

Contrairement aux tableaux statiques, il est nécessaire de spécifier le type des éléments contenus dans le vector.

Remarque : Comme les tableaux dynamiques sont des objets comme les autres, il est tout à fait possible d'écrire une fonction renvoyant un résultat de type vector, chose qui n'est pas faisable proprement avec un tableau statique. Le prototype d'une telle fonction serait par exemple :

```
#include <vector>
using namespace std;

vector<double> f();
```

Exemple

Ecrire un programme qui remplit un tableau dynamique F par les 10 premiers termes de la suite de Fibonacci puis affiche ces éléments à l'écran (en utilisant la classe vector).

La suite de Fibonacci est définie par :

- $F_0 = 1$
- $F_1 = 1$
- $F_n = F_{n-2} + F_{n-1}$ pour $n > 1$

Solution

```
#include <iostream>
#include <vector>
using namespace std;

vector<int> remplir(int n)
{
	vector<int> F (n);
	F[0] = 1;
	F[1] = 1;
	for (int i = 2; i < n; i++)
	{
		F[i] = F[i-2] + F[i-1];
	}
	return F;
}

void afficher(vector<int> F)
{
	for (int i = 0; i < F.size(); i++)
	{
		cout<<F[i]<<"\t";
	}
}

int main()
{
	vector<int> Fib = remplir(10);
	afficher(Fib);
	cout<<endl;
	return 0;
}
```

Output du programme



```
1	1	2	3	5	8	13	21	34	55
```

Exercice 1 (QCM)

1. Qu'est-ce qu'un tableau à dimension dynamique ?
   - O Un tableau dont les valeurs changent au cours du temps
   - O Un tableau qui peut avoir une taille variable
   - O Un tableau qui s'agrandit ou se rétrécit automatiquement selon les besoins

2. Quelle déclaration crée un tableau de 10 entiers dont la valeur est 3 ?
   - O `std::vector<int> tableau;`
   - O `std::vector<int> tableau(10);`
   - O `std::vector<int> tableau(10,3);`
   - O `std::vector<int> tableau(3,10);`

3. Comment trouve-t-on la taille du tableau dynamique monTab ?
   - O En écrivant `sizeof(monTab);`
   - O En écrivant `monTab.length();`
   - O En écrivant `monTab.size();`

4. Comment ajouter des éléments à la fin d'un vector ?
   - O En réallouant la mémoire avec `malloc`
   - O En réallouant la mémoire avec `new[]`
   - O En utilisant la fonction membre `push_back()`
   - O Ce n'est pas possible

5. Comment faire pour vider un vector ?
   - O En utilisant la fonction membre `empty()`
   - O En utilisant la fonction membre `clear()`
   - O En utilisant `delete[]`
   - O En supprimant les cases une-à-une avec la fonction membre `pop_back()`

6. Que contiendra le tableau à la fin de l'exécution du code suivant ?

```
std::vector<int> tableau(10,2);
tableau.resize(2);
tableau.push_back(8);
tableau.resize(6,3);
tableau.pop_back();
```

- O Il est vide.
- O 10, 10, 8, 6, 6, 6, 6, 6.
- O 2, 2, 8, 6, 6.
- O 2, 2, 8, 3, 3.
- O 10, 10, 8, 8, 8.

Exercice 2 : Ecrire un programme qui permet de créer un tableau dynamique *Temperatures* qui contiendra les températures enregistrées dans un certain nombre de villes (le nombre exact sera fourni par l'utilisateur lors de l'exécution).

Le programme doit faire appel à trois fonctions :

- la fonction `vector<int> remplir(int n)` qui permet de saisir les températures ;
- la fonction `int min(const vector<int> & T)` qui renvoie la température minimale enregistrée ;
- la fonction `int max(const vector<int> & T)` qui renvoie la température maximale enregistrée.

Solution

```
#include <iostream>
#include <vector>

using namespace std;

vector<int> remplir(int n)
{
      vector<int> T(n);
```

```
        for (int i = 0; i < n; i++)
        {
                cout<<"Temperatures["<<i<<"] = ";
                cin>>T[i];
        }
        return T;
}

int min(const vector<int> & T)
{
        int m = T[0];
        for (int i = 1; i < T.size(); i++)
        {
                if (T[i] < m)
                        m = T[i];
        }
        return m;
}

int max(const vector<int> & T)
{
        int M = T[0];
        for (int i = 1; i < T.size(); i++)
        {
                if (T[i] > M)
                        M = T[i];
        }
        return M;
}

int main()
{
        int n, m, M;
        cout<<"Entrer le nombre de villes : ";
        cin>>n;
        vector<int> Temperatures(n);
        Temperatures = remplir(n);
        m = min(Temperatures);
        M = max(Temperatures);
        cout<<"Temperature minimale : "<<m<<endl;
        cout<<"Temperature maximale : "<<M<<endl;
        cout<<endl;
        return 0;
}
```

Test du programme

```
Entrer le nombre de villes : 10

Temperatures[0] = 23
Temperatures[1] = 22
Temperatures[2] = 25
Temperatures[3] = 27
Temperatures[4] = 26
Temperatures[5] = 28
Temperatures[6] = 24
Temperatures[7] = 30
Temperatures[8] = 24
Temperatures[9] = 20

Temperature minimale : 20
Temperature maximale : 30
```

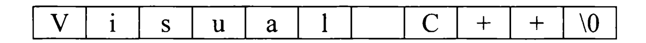
Les chaînes de caractères

1. Qu'est-ce qu'une chaîne de caractères ?

Une chaîne de caractères (appelée *string* en anglais) est une suite de caractères, c'est-à-dire un ensemble de symboles (lettres, chiffres et caractères spéciaux) faisant partie du jeu de caractères défini par le code ASCII.

En langage C++, une chaîne de caractères est un tableau comportant plusieurs données de type **char** dont le dernier élément est le caractère nul '**\0**', c'est-à-dire le premier caractère du code ASCII (dont la valeur est 0). Ce caractère est un caractère de contrôle (donc non affichable) qui permet d'indiquer la fin de la chaîne de caractères.

Ainsi, une chaîne composée de *n* caractères sera en fait un tableau de *(n+1)* éléments de type char. On peut par exemple représenter la chaîne "Visual C++" de la manière suivante :



V	i	s	u	a	l		C	+	+	\0

2. Créer une chaîne de caractères

Pour définir une chaîne de caractères en langage C++, il suffit de définir un tableau de caractères en utilisant la syntaxe suivante :

```
char  Nom_Chaîne[Nombre d'éléments] ;
```

Le nombre maximum de caractères que comportera la chaîne sera égal au nombre d'éléments du tableau moins un (réservé au caractère de fin de chaîne). On peut toutefois utiliser partiellement cet espace en insérant le caractère de fin de chaîne à l'emplacement désiré dans le tableau.

3. Initialiser une chaîne de caractères

Il est toujours conseillé d'initialiser la chaîne de caractères, c'est-à-dire remplir les cases du tableau avec des caractères, sachant que celui-ci devra **obligatoirement** se terminer par le caractère de fin de chaîne '\0'.

Il y a trois façons de procéder :

- remplir manuellement le tableau case par case ;
- affecter une chaîne de caractères au tableau lors de la déclaration ;
- utiliser les fonctions de manipulation de chaînes fournies dans les bibliothèques standards.

Exemple

```
#include <iostream>
using namespace std;

int main ( )
{
        char chaine1[4] = {'C','+','+','\0'};
        char chaine2[7] = "Visual";
        char chaine3[] = "Studio";
        return 0;
}
```

Dans cet exemple, la longueur de chaine3 n'a pas été précisée de façon explicite mais le compilateur déduit qu'il s'agit d'une chaîne de taille 7.

4. Les fonctions de manipulation de chaînes de caractères

a. La fonction *strcpy( )*

La fonction *strcpy( )* (abréviation de *string copy*) est une fonction qui permet de copier une chaîne de caractères dans une autre. Cette fonction admet comme paramètres les deux chaînes de caractères. Elle copie le contenu de la deuxième chaine dans la première et renvoie 1 si la copie s'est déroulée correctement, sinon elle renvoie 0.

La syntaxe de la fonction *strcpy( )* est la suivante :

```
strcpy(chaine_destination , "chaine de caractères");
```

ou encore :

```
strcpy(chaine_destination , chaine_source);
```

Exemple

```
#include <iostream>
using namespace std;

int main ( )
{
        char Chaine1[20];
        char Chaine2[20];
        strcpy(Chaine1, "Visual C++");
        strcpy(Chaine2, Chaine1);
        cout<<"Chaine1 = "<<Chaine1<<endl;
        cout<<"Chaine2 = "<<Chaine2<<endl;
        return 0;
}
```

Output du programme

```
Chaine1 = Visual C++
Chaine2 = Visual C++
```

b. La fonction *strcat()*

La fonction *strcat()* permet de faire la concaténation (assemblage) de deux chaînes de caractères. Le résultat sera mis dans la chaîne fournie en tant que premier paramètre.

La syntaxe de cette fonction est la suivante :

strcat(chaine1 , chaine2);

Exemple

```
#include <iostream>
using namespace std;

int main()
{
        char ch1[10] = "long";
        char ch2[6] = "temps";
        strcat(ch1, ch2);
        cout <<"ch1 = "<<ch1<<endl;
        return 0;
}
```

Output du programme

```
ch1 = longtemps
```

c. La fonction *strlen()*

La fonction *strlen()* (abréviation de *string length*) renvoie la longueur d'une chaîne de caractère (sans compter le caractère de fin de chaîne \0).

La syntaxe de cette fonction est la suivante :

strlen(chaine)

Exemple

```
#include <iostream>
using namespace std;

int main()
{
    char Nom[50];
    cout<<"C'est quoi votre nom ? : ";
    cin.getline(Nom,50);
    int L = strlen(Nom);
    cout <<"Votre nom contient "<<L<<" caracteres.";
    cout<<endl;
    return 0;
}
```

Test du programme

```
C'est quoi votre nom ? : Mohamed Ali
Votre nom contient 11 caracteres.
```

Dans le programme précédent, nous avons utilisé la fonction *cin.getline( )* pour lire le nom de l'utilisateur étant donné que ce dernier peut contenir des espaces. Le paramètre 50 indique le nombre maximal de caractères qui peuvent être entrés.

d. La fonction *strcmp( )*

La fonction *strcmp( )* (abréviation de *string compare*) est une fonction qui permet de comparer deux chaînes de caractères. En standard, il n'est pas possible d'effectuer une comparaison de chaînes de caractères avec la simple utilisation des opérateurs habituels (==, !=, >=, etc.) à moins d'utiliser une boucle et faire la comparaison caractère par caractère. Il est donc préférable d'utiliser cette fonction fournie en standard avec le C++. Cette fonction admet comme paramètres deux chaînes de caractères. Elle renvoie 0 si les deux chaînes sont identiques. Si les chaînes sont différentes, elle renvoie 1.

La syntaxe de cette fonction est la suivante :

strcmp(chaine1 , chaine2)

Exemple

```
#include <iostream>
using namespace std;

int main( )
{
   char  ch1[30] = "Visual C";
   char  ch2[30] = "Visual C++";
   if(strcmp(ch1 , ch2)==0)
      cout<<"Les 2 chaines sont identiques"<<endl;
   else
      cout<<"Les 2 chaines sont differentes"<<endl;
   return 0;
}
```

Output du programme

```
Les 2 chaines sont differentes
```

Remarques

1. En standard, C++ ne permet pas d'affecter une chaîne de caractères à une autre après la déclaration, pour ce faire, il faut utiliser la fonction *strcpy( )*.

2. En C++, les caractères doivent être mis entre des quottes simples alors que les chaînes de caractères doivent être mises entres des quottes doubles.

Exemple

```
char  c = 'A';
char  ch[4] = "C++";
```

Exercice : Ecrire un programme qui lit une phrase puis compte le nombre de mots dans cette phrase. On suppose que la phrase commence obligatoirement par une lettre et que les mots sont séparés par des espaces simples.

Solution

Nombre de mots = nombre d'espaces + 1

```
#include <iostream>
using namespace std;

int NbMots(char st[])
{
	int nb = 0;
	int i = 0;
	while (st[i] != '\0')
	{
		if (st[i] == ' ')
			nb++;
		i++;
	}
	return (nb+1);
}

int main()
{
	char phrase[100];
	cout<<"Enter une phrase : ";
	cin.getline(phrase,100);
	int L = NbMots (phrase);
	cout <<"Cette phrase contient "<<L<<" mots.";
	cout<<endl;
	return 0;
}
```

Test du programme

```
Enter une phrase : oeuil pour oeuil, dent pour dent
Cette phrase contient 6 mots.
```

5. Utilisation de la classe <string>

La notion de classe est une notion très importante en C++ pour laquelle nous avons réservé un chapitre particulier dans la suite du livre (voir page 157).

Le type d'objet <string> est livré dans la librairie standard du C++. Pour l'utiliser il faut inclure le fichier d'en-tête <string> au début du programme grâce à la directive :

```
#include <string>
```

Le fichier <string> contient les prototypes de nombreuses fonctions permettant de simplifier la création et la manipulation des chaînes de caractères.

a. Affectation d'une chaîne à une variable

Pour affecter une chaîne à un objet de type string au moment de la déclaration, il y a plusieurs possibilités. La plus courante consiste à ouvrir des parenthèses comme si on appelait une fonction :

string NomChaine("Chaîne de caractères") ;

Il est également possible d'affecter une chaîne à un objet de type string au moment de la déclaration ou après en utilisant l'opérateur d'affectation (=).

Exemple

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
        string chaine1("Visual C");
        string chaine2 = "Visual C++";
        chaine1 = chaine2;
        cout<<"chaine1 = "<<chaine1<<endl;
        cout<<"chaine2 = "<<chaine2<<endl;
        return 0;
}
```

Output du programme

```
chaine1 = Visual C++
chaine2 = Visual C++
```

Dans le programme précédent, nous n'avons pas précisé la longueur de chaine1, puis nous avons affecté à cette variable une nouvelle chaîne plus grande qu'avant sans que le compilateur ne signale aucune erreur. En effet, des mécanismes sont prévus pour redimensionner automatiquement ce type de variables en fonction de la longueur de la chaîne à stocker dedans. C'est bien là tout l'intérêt de la Programmation Orientée Objet : l'utilisateur n'a pas besoin de comprendre comment ça marche à l'intérieur. L'objet est en quelque sorte intelligent et gère tous les cas.

b. Concaténation de chaînes

Avec la classe <string>, la concaténation de 2 chaînes se fait facilement en utilisant l'opérateur « + ».

Exemple

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string chaine1 = "Microsoft";
    string chaine2 = "Visual C++";
    string chaine3;
    chaine3 = chaine1 + " " + chaine2;
    cout<<"chaine3 = "<<chaine3<<endl;
    return 0;
}
```

Output du programme

```
chaine3 = Microsoft Visual C++
```

c. Comparaison de chaînes

Avec la classe <string>, on peut comparer des chaînes entre elles à l'aide des symboles == et != que nous avons utilisés pour comparer des variables de type simple.

Exemple

```
#include <iostream>
#include <string>
using namespace std;

int main( )
{
   string ch1 = "Visual C";
   string ch2 = "Visual C++";
   if(ch1 == ch2)
      cout<<"Les 2 chaines sont identiques"<<endl;
   else
      cout<<"Les 2 chaines sont differentes"<<endl;
   return 0;
}
```

Output du programme

```
Les 2 chaines sont differentes
```

d. Quelques fonctions membres de la classe string

- ***La méthode size( )***

La méthode *size( )* permet de connaître la longueur de la chaîne actuellement stockée dans l'objet de type string. C'est un peu l'équivalent de *strlen( )*, mais cette méthode ne prend aucun paramètre.

Exemple

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
   string Nom = "Mohamed Ali";
   int L = Nom.size();
   cout <<"Ce nom contient "<<L<<" caracteres." ;
   cout<<endl;
   return 0;
}
```

Output du programme

```
Ce nom contient 11 caracteres.
```

- ***La méthode erase( )***

Cette méthode supprime tout le contenu de la chaîne actuelle.

Exemple

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
     string chaine = "Visual C++";
     chaine.erase();
     cout <<"La chaine contient : "<<chaine<<endl;
     return 0;
}
```

Output du programme

```
La chaine contient :
```

- ***La méthode substr( )***

Le mot substr signifie "substring", soit "sous-chaîne" en anglais.

Cette méthode permet d'extraire une partie de la chaîne de caractères stockée dans un objet de type string.

Sa syntaxe est la suivante :

NomChaine.substr(index, npos)

Cette méthode prend deux paramètres :

- *index* qui permet d'indiquer à partir de quel caractère on doit couper (ce doit être un numéro de caractère).
- *npos* qui permet d'indiquer la longueur de la sous-chaîne à extraire. Par défaut, la fonction renvoie tous les caractères qui restent.

Exemple

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
     string chaine = "Visual C++";
     string schaine1 = chaine.substr(0,4);
     string schaine2 = chaine.substr(7);
     cout<<"schaine1 = "<<schaine1<<endl ;
     cout<<"schaine2 = "<<schaine2<<endl ;
     return 0;
}
```

Output du programme

```
schaine1 = Visu
schaine2 = C++
```

Exercice 1 (QCM)

1. Laquelle de ces déclarations d'un objet de type string est **incorrecte** ?
   - O `string chaine(Ma chaine);`
   - O `string chaine = "Ma chaine";`
   - O `string chaine("Ma chaine");`

2. Que vaudra maChaine après ces opérations ?

```
string maChaine;
maChaine = "Week";
maChaine += "end";
```

   - O Week
   - O end
   - O Weekend
   - O Week end

3. Peut-on afficher un string à l'aide d'un cout ?
   - O Oui
   - O Non
   - O Oui, mais à condition de le convertir en char* avec c_str( )

4. Que va afficher ce code ?

```
string message("Programme");
cout << message.substr(7);
```

   - O Program
   - O me
   - O Programme

Exercice 2 : Ecrire un programme qui contient une fonction Palind( ) dont l'en-tête est de la forme :

```
bool Palind(string s)
```

et qui vérifie si une chaîne de caractères est un palindrome ou non.

Un palindrome est un mot qui peut être lu indifféremment de droite à gauche ou de gauche à droite.

Exemples : "AZIZA", "LAVAL", "RADAR", "2002".

Solution

```
#include <iostream>
#include <string>
using namespace std;

bool Palind(string s)
{
	int i;
	int n = s.size();
	for(i = 0; i < n/2; i++)
	{
		if (s[i] != s[n-i-1])
			return false;
	}
	return true;
}

int main()
{
	string st;
	cout<<"Entrer une chaine : ";
	cin>>st;
	if (Palind(st))
		cout<<"C'est un palindrome"<<endl;
	else
		cout<<"Ce n'est pas un palindrome"<<endl;
	return 0;
}
```

Test du programme

```
Entrer une chaine : LAVAL
C'est un palindrome
```

EXERCICES D'APPLICATION

Exercice 1 : Ecrire un programme qui lit une chaîne de caractères puis affiche son inverse.

A titre d'exemple, si la chaîne entrée est "algo", le programme doit afficher "ogla".

Exercice 2 : Ecrire un programme qui lit une chaîne de caractères et renvoie son équivalent en majuscule.

Note : la fonction *toupper( )* permet de convertir un caractère quelconque en majuscules.

Exercice 3 : Ecrire un programme qui détermine et affiche le mot le plus long dans une phrase donnée.

Exercice 4 : Ecrire un programme qui lit :

- Un mot (chaîne de caractères formée uniquement de lettres)
- Une lettre

puis affiche le nombre d'apparitions de la lettre dans le mot.

Les pointeurs

1. Définition d'un pointeur

Un pointeur est une variable contenant l'adresse d'une autre variable d'un type donné. La notion de pointeur fait souvent peur car il s'agit d'une technique de programmation très puissante, permettant de définir des structures dynamiques, c'est-à-dire qui évoluent au cours du temps (par opposition aux tableaux statiques par exemple qui sont des structures de données dont la taille est figée à la définition).

2. La notion d'adresse

Comme nous l'avons vu, un pointeur est une variable qui permet de stocker une adresse, il est donc nécessaire de comprendre ce qu'est une adresse.

Lorsque l'on exécute un programme, celui-ci est chargé en mémoire, cela signifie que d'une part le code à exécuter est stocké, mais aussi que chaque variable déclarée possède une zone de mémoire qui lui est réservée, la taille de cette zone dépond du type de variable.

En réalité, la mémoire est constituée de plein de petites cases (ou cellules) de 8 bits (1 octet). Une variable, selon son type (donc sa taille), va ainsi occuper une ou plusieurs de ces cases (une variable de type char occupera une seule case, tandis qu'une variable de type long occupera 4 cases consécutives).

Chacune de ces « cases » (appelées **blocs**) est identifiée par un numéro. Ce numéro s'appelle **adresse**.

On peut donc accéder à une variable de 2 façons :

1. grâce à son nom
2. grâce à l'adresse de la première case allouée à la variable.

Il suffit donc de stocker l'adresse de la variable dans un pointeur afin de pouvoir accéder à celle-ci (on dit que l'on « pointe vers la variable »).

La figure 9 montre par quel mécanisme il est possible de faire pointer une variable (de type *pointeur*) vers une autre. Ici le pointeur stocké à l'adresse 24 pointe vers une variable stockée à l'adresse 253 (les valeurs sont bien évidemment arbitraires).

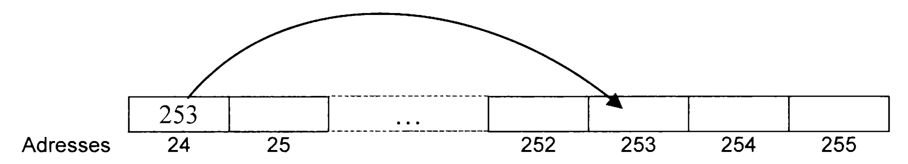


Figure 9 : Relation entre pointeur et variable pointée

3. Comment connaît-on l'adresse d'une variable ?

En réalité, on n'aura jamais à écrire l'adresse d'une variable, d'autant plus qu'elle change à chaque lancement de programme étant donné que le système d'exploitation alloue les blocs de mémoire qui sont libres, et ceux-ci ne sont pas les mêmes à chaque exécution.

Pour connaître l'adresse d'une variable connaissant son nom, il suffit de faire précéder le nom de la variable par le caractère ***&*** (« ET commercial ») de la façon suivante:

&Nom_Variable

En C++, les adresses sont toujours exprimées en hexadécimal.

Exemple

```
#include <iostream>
using namespace std;

int main()
{
	float x = 3.66;
	cout<<"Adresse de la variable x = "<<&x<<endl;
	return 0;
}
```

Output du programme

```
Adresse de la variable x = 0012FF7C
```

4. Intérêt des pointeurs

Les pointeurs ont un grand nombre d'intérêts :

- Ils permettent de manipuler de façon simple des données pouvant être importantes. A titre d'exemple, au lieu de passer à une fonction un élément de grande taille, on pourra lui fournir un pointeur vers cet élément.

- Les tableaux ne permettent de stocker qu'un nombre fixé d'éléments de même type. En stockant des pointeurs dans les cases d'un tableau, il sera possible de stocker des éléments de taille diverse, et même de rajouter des éléments au tableau en cours d'utilisation.

- Il est possible de créer des structures chaînées, c'est-à-dire comportant des maillons (voir page 203).

5. Déclaration d'un pointeur

Un pointeur est une variable qui doit être définie en précisant le type de variable pointée, de la façon suivante :

```
type * Nom_Pointeur ;
```

Le type de variable pointée peut être aussi bien un type primaire (tel que *int*, *char...*) qu'un type complexe (tel que *struct*).

Grâce au symbole '*', le compilateur sait qu'il s'agit d'une variable de type *pointeur* et non d'une variable ordinaire. Il sait également combien de blocs se situent à l'adresse pointée (d'après le type de la variable).

6. Initialisation d'un pointeur

Après avoir déclaré un pointeur il faut l'initialiser. Cette démarche est très importante car lorsqu'on déclare un pointeur, celui-ci contient ce que la case où il est stocké contenait avant, c'est-à-dire n'importe quel nombre. Autrement dit, si on n'initialise pas un pointeur, celui-ci risque de pointer vers une zone hasardeuse de la mémoire qui peut être un morceau du programme ou du système d'exploitation.

Un pointeur non initialisé représente un danger !

Pour initialiser un pointeur, il faut utiliser l'opérateur d'affectation '=' suivi de l'opérateur d'adresse '&' auquel est accolé un nom de variable (celle-ci doit bien sûr avoir été définie avant) :

Nom_Pointeur = &Nom_Variable_Pointée;

Exemple

```
int a = 2;
char c;
int *pa = &a;
char *pc = &c;
```

7. Accès à une variable pointée

Après (et seulement après) avoir déclaré et initialisé un pointeur, il est possible d'accéder au contenu de l'adresse mémoire pointée par le pointeur grâce à l'opérateur '*'. La syntaxe est la suivante :

***pointeur**

Exemple

```
*pa = 10;
*pc = 'a';
```

Après ces deux instructions, le contenu des zones mémoires pointées par *pa* et *pc* sera respectivement 10 et 97 (le code ASCII du caractère 'a').

Exercice

Que vaudront les variables i et j après l'exécution du programme suivant :

```
#include <iostream>
using namespace std;

int main( )
{
      int i = 3, j;
      int *p = &i;
      j = (*p) + 5;       // identique à j = i + 5
      (*p) = j + 2 ;      // identique à i = j + 2
```

```
        (*p)++;                // identique à i++
        cout << "i = " << i << "\t"<<"j = " <<j;
        cout<<endl;
        return 0;
}
```

Output du programme

```
i = 11   j = 8
```

8. Passage d'argument à une fonction par adresse (par pointeur)

Au lieu du passage par référence, il est possible de passer *les adresses des arguments* à une fonction et de modifier les données contenues à ces adresses via l'opérateur '*'.

Exemple

```
#include <iostream>
using namespace std;

void permut(int * a, int * b)
{
        int temp = (*a);
        (*a)= (*b);
        (*b)= temp;
}

int main()
{
        int i = 5, j = 10;
        permut(&i,&j);
        cout << "i = " << i << "\t"<<"j = " <<j;
        cout<<endl;
        return 0;
}
```

Output du programme

```
i = 10   j = 5
```

9. Tableaux et pointeurs

Un tableau est une suite d'éléments de même type définie selon la syntaxe suivante :

type NomTab[taille] ;

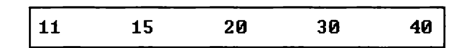
En fait, l'identificateur NomTab est un pointeur sur la première case du tableau, c'est-à-dire qu'il contient l'adresse du premier élément du tableau (&NomTab[0]). De même, l'expression NomTab[i] est équivalente à l'expression *(NomTab+i).

Exemple

```
#include <iostream>
using namespace std;

int main()
{
        int T[5] = {10, 15, 20, 30, 40};
        int * P = T;
        *P = 11;
        for (int i = 0; i < 5; i++)
              cout<<*(P + i)<<"\t";
        cout<<endl;
        return 0;
}
```

Output du programme



```
11        15        20        30        40
```

10. Les opérateurs new et delete

L'opérateur ***new*** permet d'allouer une ou plusieurs cases mémoire de façon dynamique (pendant l'exécution du programme). Cette zone mémoire peut être ensuite libérée en utilisant l'opérateur ***delete***.

- Pour allouer un espace mémoire capable de contenir un objet, on utilise la syntaxe :

Pointeur = new type ;

- Pour allouer un espace mémoire capable de contenir N objets de même type (un tableau), on utilise la syntaxe :

Pointeur = new type[N];

- Pour libérer l'espace alloué, on utilise la syntaxe :

delete Pointeur ;

ou

delete[] Pointeur ;

Exemple

```
#include <iostream>
using namespace std;

int main()
{
	int  *p;
	p = new int;
	*p = 4264;
	cout<<"*p = "<<*p<<endl;
	delete p;
	return 0;
}
```

Output du programme

```
*p = 4264
```

EXERCICES D'APPLICATION

Exercice 1 : Qu'affichera le programme suivant si l'on suppose que l'adresse de la variable n est 0012FF7C?

```
#include <iostream>
using namespace std;

int main()
{
	int n = 99;
	int *ptr = &n;
	cout<<n<<"\t"<<*ptr<<"\t"<<ptr;
	cout<<endl;
	return 0;
}
```

Exercice 2 : Ecrire un programme permettant de :

- définir une variable réelle x de valeur initiale 95.5
- définir un pointeur p de type réel
- faire pointer p vers la variable x
- ajouter 4.5 à l'emplacement mémoire pointé par p
- afficher la valeur de la variable x ainsi que son adresse en mémoire.

Exercice 3 : Qu'affiche le programme suivant :

```
#include <iostream>
using namespace std;

int main ( )
{
	int A[ ] = {10, 20, 30, 40, 50};
	int *ptr = A;
	cout<<A[3]<<"\t"<<*(ptr+3)<<"\t"<<ptr[3];
	cout<<endl;
	return 0;
}
```

Les structures

1. Notion de structure

Contrairement aux tableaux qui permettent de regrouper des éléments de même type, c'est-à-dire codés sur le même nombre de bits et de la même façon, les structures permettent de rassembler des éléments de type différent au sein d'une seule entité repérée par un nom. Les objets contenus dans la structure sont appelés **champs** de la structure.

A titre d'exemple, une date, une adresse ou un nombre complexe peuvent être considérés comme des structures.

2. Déclaration d'une structure

Lors de la déclaration, il faut indiquer les champs de la structure, c'est-à-dire le type et le nom des attributs qui la composent :

```
struct  Nom_Structure
{
        type1 Champ1;
        type2 Champ2;
        ...
};
```

- La dernière accolade doit être suivie d'un point-virgule ;
- Le nom des champs doit répondre aux critères des noms de variable ;
- Les données peuvent être de n'importe quel type hormis le type de la structure dans laquelle elles se trouvent.

Exemple

```
struct Adresse
{
        string Rue;
        string Ville;
        int CodePostal;
};
```

```
struct Personne
{
      string Nom;
      int Age;
      Adresse Adr;
};
```

3. Définition d'une variable structurée

La déclaration d'une structure ne fait que donner l'allure de la structure, c'est-à-dire en quelque sorte la définition d'un type de variable complexe. La déclaration ne réserve pas d'espace mémoire, il faut donc définir une (ou plusieurs) variable(s) structurée(s) après avoir déclaré le type structure.

La définition d'une variable structurée se fait comme suit :

Nom_Structure Nom_Variable;

- *Nom_Structure* représente le nom d'une structure que l'on a préalablement déclarée ;
- *Nom_Variable* est le nom que l'on donne à la variable de type structure.

Il va de soi que, comme dans le cas des autres variables, on peut définir plusieurs variables structurées en les séparant avec des virgules :

Nom_Structure Nom_Var1, Nom_Var2, ...;

Exemple

```
Personne P1, P2;
```

4. Accès aux champs d'une variable structurée

Chaque variable de type structure possède des champs repérés par des noms uniques. Pour accéder aux champs d'une structure, on utilise l'opérateur de champ (un simple point) placé entre le nom de la variable structurée que l'on a définie et le nom du champ :

Nom_Variable.Nom_Champ

Ainsi, pour affecter des valeurs à la variable P1 (variable de type *Personne* définie précédemment), on pourra écrire :

```
P1.Nom = "Ali";
P1.Age = 21;
P1.Adr.Rue = "Ibn Rochd";
P1.Adr.Ville = "Monastir";
P1.Adr.CodePostal = 5000;
```

Remarque

En C++, il est possible d'affecter une variable structurée à une autre variable du même type, comme dans l'exemple suivant :

```
                  P2.Nom = P1.Nom;
P2 = P1;  ⇔     { P2.Age = P1.Age;
                  P2.Adr = P1.Adr;
```

Exercice

Ecrire un programme qui lit deux nombres complexes C1 et C2 et qui affiche ensuite leur somme et leur produit.

On utilisera les formules de calcul suivantes :

- $(a + bi) + (c + di) = (a + c) + (b + d)i$
- $(a + bi)\,(c + di) = (ac - bd) + (ad + bc)i$

Solution

```
#include <iostream>

using namespace std;

struct complex
{
   float Reel;
   float Imag;
};

complex somme(complex n1, complex n2)
{
   complex n3;
```

```
    n3.Reel = n1.Reel + n2.Reel;
    n3.Imag = n1.Imag + n2.Imag;
    return n3;
}

complex produit(complex n1, complex n2)
{
    complex n3;
    n3.Reel = n1.Reel*n2.Reel - n1.Imag*n2.Imag;
    n3.Imag = n1.Reel*n2.Imag + n1.Imag*n2.Reel;
    return n3;
}

int main()
{
    complex c1, c2, c3,c4;
    c1.Reel = 2;
    c1.Imag = 2;
    c2.Reel = 1;
    c2.Imag = 3;
    c3 = somme(c1,c2);
    c4 = produit(c1,c2);
    cout<<"c1 = "<<c1.Reel<<" + "<<c1.Imag<<"i";
    cout<<endl;
    cout<<"c2 = "<<c2.Reel<<" + "<<c2.Imag<<"i";
    cout<<endl;
    cout<<"c3=c1+c2="<<c3.Reel<<"+"<<c3.Imag<<"i";
    cout<<endl;
    cout<<"c4=c1*c2="<<c4.Reel<<"+"<<c4.Imag<<"i";
    cout<<endl;
    return 0;
}
```

Output du programme

```
c1 = 2 + 2i
c2 = 1 + 3i
c3 = c1 + c2 = 3 + 5i
c4 = c1 * c2 = -4 + 8i
```

Remarque

Le langage C++ permet d'initialiser les variables structurées à la déclaration comme dans l'exemple suivant :

```
struct complex
{
        float Reel;
        float Imag;
};

complex c1 = {2,1};         //c1 = 2 + i
complex c2 = {1,3};         //c2 = 1 + 3i
```

5. Pointeurs de structure

Comme les autres types de données, C++ permet de créer des pointeurs sur des variables de type structure. L'accès aux champs de la variable par le pointeur se fera à l'aide de l'opérateur '->' :

Pointeur -> Champ

ou via l'opérateur de déréférencement '*' :

***(Pointeur).Champ**

Exemple

```
#include <iostream>

using namespace std;

int main()
{
   struct Etudiant
   {
      int Num ;
      char Nom[20] ;
      char Prenom[20] ;
      char Classe[6] ;
   } ;
   Etudiant e = {1001, "Ben Salah", "Ali", "Info1"};
```

```
    Etudiant *p = &e;
    strcpy(p -> Classe,"Info2") ;
    cout<<e.Nom<<"\t"<<e.Prenom<<"\t"<<e.Classe ;
    cout<<endl ;
    return 0 ;
}
```

Output du programme

```
Ben Salah        Ali      Info2
```

6. Tableaux de structures

Etant donné qu'une structure est composée d'éléments de taille fixe, il est possible de créer un tableau contenant des éléments de type structure en utilisant la syntaxe suivante :

Nom_Structure Nom_Tableau[Nbre d'éléments];

A titre d'exemple, le tableau suivant (nommé *Repertoire*) pourra contenir 80 variables structurées de type *Personne* :

```
Personne Repertoire[80];
```

Pour accéder aux champs d'une personne particulière, on pourra écrire par exemple :

```
Repertoire[0].Nom = "Ahmed" ;
Repertoire[0].Age = 20 ;
Repertoire[0].Adr.Rue = "Ibn Sina" ;
...
```

Exercice

Créer un tableau TabEmp qui contiendra les informations sur les 50 employés d'une entreprise (Matricule, Nom, Salaire), le remplir puis afficher la liste des employés sous forme d'un tableau.

Solution

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

struct Employe
{
   int  Matricule;
   string Nom;
   float Salaire;
};

void remplir(vector<Employe> & T)
{
   int i;
   for (i = 0; i < T.size(); i++)
   {
      cout<<"Entrer le matricule : ";
      cin>>T[i].Matricule;
      cout<<"Entrer le nom : ";
      cin>>T[i].Nom;
      cout<<"Entrer le Salaire : ";
      cin>>T[i].Salaire;
      cout<<"--------------------------"<<endl;
   }
}

int main( )
{
   vector<Employe> TabEmp(50);   // Création du tableau
   int i;
   remplir(TabEmp);  // Remplissage du tableau
   cout<<"MATRICULE\tNOM\tSALAIRE\n";
   cout<<"---------\t---\t-------\n";
   for (i = 0; i < TabEmp.size(); i++)
   {
      cout<<TabEmp[i].Matricule<<"\t"<<TabEmp[i].Nom;
      cout<<"\t"<<TabEmp[i].Salaire;
      cout<<endl;
   }
   return 0;
}
```

EXERCICES D'APPLICATION

Exercice 1

Créer un tableau *Livres* qui contiendra les informations sur les livres disponibles dans une bibliothèque (Numéro, Titre, Auteur, Thème, Nombre d'exemplaires), puis afficher la liste des livres d'informatique sous forme d'un tableau.

Exercice 2

Créer un tableau *Comptes* qui contiendra les informations sur les comptes bancaires (Numéro du compte, Nom du client, Date d'ouverture, Solde), puis afficher la liste des comptes débiteurs (solde < 0).

Exercice 3

Créer un tableau *Etudiants* qui contiendra les informations sur les étudiants d'une classe avec leur notes en programmation (Numéro de l'étudiant, Nom, Prénom, Note) puis afficher la liste des étudiants qui ont obtenu une note supérieure à 12.

Exercice 4

Ecrire un programme pour un petit commerçant qui vend des produits agro-alimentaires. Chaque produit est caractérisé par un numéro, un libellé, un prix de vente et une quantité en stock.

Le programme doit permettre au commerçant de gérer son stock à partir d'un menu qui a l'allure suivante :

```
==================================
          GESTION DU STOCK
1-  Consulter la liste des produits
2-  Enregistrer un mouvement (entrée/sortie)
3-  Ajouter un nouveau produit
4-  Chercher un produit (à partir du code)
5-  Supprimer un produit
6-  Quitter le programme
==================================
Entrer votre choix (1,2,3,4,5,6) : _
==================================
```

La Programmation Orientée Objet

Introduction

Le langage C++ a été principalement inventé pour supporter la Programmation Orientée Objet.

La POO est un paradigme ou approche de programmation qui consiste à structurer le code source en un ensemble de ***classes***. Une classe est constituée :

- de variables membres, appelées ***attributs***
- de fonctions membres appelées ***méthodes***

La POO n'est pas utilisée qu'en C++. De nombreux langages, encore plus récents, exploitent au maximum les concepts de la POO (C#, Java, Python, Smalltalk, Visual Basic.Net, …).

1. Origines

Le langage Simula-67, en implantant les Record Class, a posé les constructions qui seront celles des langages orientés objet à classes : classe, polymorphisme, héritage, etc. Mais c'est réellement avec le langage Smalltalk, inspiré en grande partie de Simula 67 et de Lisp, que les principes de la programmation par objets, résultat des travaux d'Alan Kay, sont diffusés : objet, encapsulation, messages, typage et polymorphisme (via la sous-classification) ; les autres principes, comme l'héritage, sont soit dérivés de ceux-ci ou une implantation. Dans Smalltalk, tout est objet, même les classes.

L'effervescence des langages à objets a commencé à partir des années 1980 : Objective C, C++ (*C with classes*) en 1983, Eiffel, Common Lisp Object System, etc. Les années 1990 voient l'âge d'or de l'extension de la programmation par objet dans les différents secteurs du développement logiciel.

Depuis, la programmation par objet n'a cessé d'évoluer aussi bien dans son aspect théorique que pratique et différents métiers et concepts ont vu le jour :

- l'analyse objet (AOO) ;
- la conception objet (COO) ;
- les bases de données objet et les SGBDOO ;
- les langages à prototypes ;
- etc.

Aujourd'hui, la programmation par objet est vue davantage comme un paradigme, le **paradigme objet**, que comme une simple technique de programmation. C'est pourquoi, lorsque l'on parle de nos jours de programmation par objets, on désigne avant tout la partie codage d'un modèle à objets obtenu par AOO et COO.

2. La notion d'objet

L'idée de la programmation orientée objet, c'est de manipuler des éléments que l'on appelle des "***objets***" dans son code source.

Un objet est constitué d'un ensemble de données appelées ***attributs*** et de fonctions membres appelées ***méthodes***.

En théorie, on peut accéder aux variables membres (les attributs) de l'objet de la même manière qu'on le faisait avec les structures. Cependant, en POO, il y a une règle super importante appelée ***règle d'encapsulation*** qui exige que tous les attributs d'une classe doivent toujours être privés (cachés). L'utilisateur ne doit pouvoir accéder qu'aux fonctions membres qui lisent et modifient les attributs de l'objet pour le faire évoluer.

3. La notion de classe

Pour créer un objet, il faut d'abord créer une ***classe***. En effet, chaque objet est une instance d'une classe. Cela signifie qu'un objet est la matérialisation concrète d'une classe (tout comme la maison qui est la matérialisation concrète du plan de la maison).

A titre d'exemple, la classe voiture peut être représentée par le schéma suivant :

Voiture
Attributs - Modèle - Année - Couleur
Méthodes - Démarrer( ) - Arrêter( ) - Accélérer(int vitesse)

L'objet MaVoiture qui est une instance de la classe Voiture peut être représenté comme suit :

MaVoiture
Modèle : Renault Mégane Année : 2007 Couleur : Gris

Avant la création des classes et des objets, il faut passer par une étape de réflexion dans laquelle il faut ressortir la liste des classes à définir ainsi que les attributs et les méthodes de chaque classe. Il est important de savoir qu'un langage spécial appelé UML (Unified Modeling Language) a été spécialement conçu pour "dessiner" les classes avant de commencer à les coder.

4. Les avantages de POO

La POO consiste entre autres, à concevoir un programme, un script ou une application dans l'espace plutôt que dans le temps. Grosso-modo, il s'agit de penser en termes d'objets, capables d'interagir entre eux, et pas de lignes de codes qui s'enchainent les unes après les autres.

La POO offre de nombreux avantages, parmi lesquels on peut citer :

- la réutilisabilité du code source ;
- la simplification de la maintenance ;
- l'accroissement de la stabilité des programmes ;
- l'amélioration de la productivité des programmeurs ;
- l'encouragement du travail collaboratif.

En résumé, la POO offre un cadre de travail efficace et encourage la réflexion en soulageant le programmeur de fastidieuses tâches de codage.

5. La modélisation objet

La modélisation objet consiste à créer un modèle informatique du système d'information de l'utilisateur. Ce modèle peut rassembler aussi bien des éléments du monde réel que des concepts ou des idées propres au métier ou au domaine duquel fera partie le système. La modélisation objet consiste à définir et à qualifier dans un premier temps ces éléments sous forme de types, donc indépendamment de leur mise en œuvre. C'est ce que l'on appelle l'*analyse orientée objet* (Object-Oriented Analysis). Puis, on propose une solution technique pour représenter les éléments définis dans le modèle d'analyse. C'est ce que l'on appelle la *conception orientée objet* (Object-Oriented Design).

Une fois un modèle de conception établi, il est possible au développeur de l'implémenter dans un langage de programmation orienté objet.

Pour développer ces différents modèles, différents langages et méthodes ont été mis au point, dont OMT de Rumbaugh, BOOCH'93 de Booch et OOSE de Jacobson. Toutefois, ces méthodes ne permettaient de modéliser que certains types d'applications et se trouvaient limitées dans d'autres contextes.

À partir de 1994, Rumbaugh, Booch et Jacobson ont décidé de s'unir dans l'élaboration d'une nouvelle méthode suffisamment générique pour pouvoir s'appliquer à quasiment tous les contextes applicatifs. Ils ont commencé d'abord par définir un langage de modélisation fortement inspiré des méthodes des trois auteurs : *UML* (Unified Modeling Language). Une fois celui-ci est pris en charge par l'OMG (Object Management Group), Rumbaugh, Booch et Jacobson se sont attaqués à la méthode proprement dite : *USDP* (Unified Software Development Process). Cette méthode définit un cadre générique de développement objet avec UML comme langage de modélisation. *USDP* est une méthode itérative et incrémentale, centrée sur l'architecture et guidée par les cas d'utilisation et la réduction des risques.

Exercice (QCM)

1. Le langage C++ a été inventé parce que le C n'était pas assez portable.
   - O Vrai
   - O Faux

2. Parmi les avantages de la POO
   - O Elle permet de mieux organiser le code source
   - O Elle permet de développer des programmes plus rapides
   - O Elle permet de développer des programmes open source

3. Quel est le rapport entre un objet et une classe ?
   - O Une classe est une instance d'objet
   - O Un objet est une instance de classe
   - O Il n'y a aucun rapport

4. Qu'est-ce qu'une méthode ?
   - O Une variable contenue dans un objet
   - O Une fonction contenue dans un objet
   - O Un type d'objet particulier

5. Que dit la règle de l'encapsulation ?
   - O Toutes les méthodes d'une classe doivent être privées
   - O Tous les attributs d'une classe doivent être publics
   - O Tous les attributs d'une classe doivent être privés

6. Qu'est-ce que UML ?
   - O Un langage de programmation par objet
   - O Une méthode de conception orientée objet
   - O Un langage de modélisation objet basé sur des diagrammes

Les classes

1. Définition d'une classe

Le langage C est un langage ***procédural***, c'est-à-dire que c'est un langage permettant de définir des données grâce à des variables et des traitements grâce à des fonctions.

L'apport principal du langage C++ par rapport au langage C est l'intégration du concept objet, afin d'en faire un *langage orienté objet* (*LOO*).

Pour pouvoir manipuler des objets, il faut définir des classes, c'est-à-dire définir la structure de ces objets. En C++, la définition d'une classe se fait de la manière suivante :

```
class Nom_Classe
{
   private :
         // définition des données membres de la classe

   public :
         // définition des fonctions membres de la classe
};
```

Nom_classe représente le type d'objet désigné par la classe.

Les mots clé *private* et *public* permettent de définir le *niveau de visibilité* des éléments de la classe. Cela correspond au concept d'*encapsulation*, un des concepts majeurs de la programmation orientée objet.

Le *point-virgule* situé à la fin du bloc de définition de la classe est **obligatoire**.

2. Déclaration des données membres

Jusqu'ici la classe que nous avons définie est vide (elle est toutefois syntaxiquement correcte), c'est-à-dire qu'elle ne contient ni données (*attributs*) ni traitements (*méthodes*).

Les données membres sont des variables stockées au sein d'une classe. Elles doivent être précédées de leur type et d'une étiquette précisant leur portée, c'est-à-dire les classes ayant le droit d'y accéder.

Ces étiquettes sont en étroite relation avec le mécanisme d'encapsulation qui consiste à rassembler les données et les méthodes au sein d'une structure en cachant l'implémentation de l'objet, c'est-à-dire en empêchant l'accès aux données par un autre moyen que les services proposés. L'encapsulation permet donc de garantir l'intégrité des données contenues dans l'objet.

L'encapsulation permet de définir les niveaux de visibilité des éléments de la classe. Ces niveaux de visibilité définissent les droits d'accès aux données selon que l'on y accède par une méthode de la classe elle-même, d'une classe héritière, ou bien d'une classe quelconque. Il existe trois niveaux de visibilité :

- **Publique (public)** : les fonctions de toutes les classes peuvent accéder aux données ou aux méthodes d'une classe définies avec le niveau de visibilité *public*. Il s'agit du plus bas niveau de protection des données.
- **Protégée (protected)** : l'accès aux données est réservé aux fonctions membres de la classe et celles des classes dérivées.
- **Privée (private)** : l'accès aux données est limité aux méthodes de la classe elle-même. Il s'agit du niveau de protection des données le plus élevé.

Ainsi, la déclaration de la classe Personne comportant trois données membres (âge, taille et sexe) peut se faire comme suit :

```
class Personne
{
   private:
       int _age;
       float _taille;
       char _sexe;
 };
```

Remarque : Par défaut, tous les éléments d'un objet sont privés (private).

3. Déclaration des fonctions membres

Les données membres permettent de conserver les informations relatives à la classe, tandis que les fonctions membres représentent les traitements qu'il est possible de réaliser avec les objets instanciés de la classe. On parle généralement de *fonctions membres* ou *méthodes* pour désigner ces traitements.

Il existe deux façons pour définir les fonctions membres :

- En définissant le prototype et le corps de la fonction à l'intérieur de la classe en une seule opération ;
- En définissant le prototype de la fonction à l'intérieur de la classe et le corps à l'extérieur.

C'est cette seconde solution qui est généralement utilisée. Ainsi, puisque l'on définit la fonction membre à l'extérieur de sa classe, il est nécessaire de préciser de quelle classe cette fonction fait partie. On utilise pour cela l'**opérateur de résolution de portée** (ORP), noté ::. Il suffit ainsi de faire précéder le nom de la fonction par le nom de la classe, suivi par l'ORP pour lever toute ambiguïté (deux classes peuvent avoir des fonctions membres portant le même nom).

Voici donc la syntaxe de définition des fonctions membres :

```
class  Nom_Classe
{
   private :
         // définition des données membres de la classe

   public :
         type  Nom_Fonction(type_param1, type_param2,...);

};
type  Nom_Classe::Nom_Fonction(param1, param2, ...)
{
         // Instructions composant le corps de la fonction
}
```

Exemple

```
class Rectangle
{
   private:
       float Longueur;
       float Largeur;

   public:
       float Perimetre();   //  calul du périmetre
       float Surface();     // calcul de la surface
};

float Rectangle::Perimetre()
{
       return 2*(Longueur+Largeur);
}

float Rectangle::Surface()
{
       return Longueur*Largeur;
}
```

Etant donné que l'encapsulation doit permettre la protection des données membres (qui sont alors précédées de l'étiquette *private* ou *protected*), les fonctions membres doivent pouvoir servir d'interface pour manipuler les données membres. On place donc l'étiquette *public* devant les fonctions membres dédiées à la manipulation des données membres. Ce système permet de garantir l'intégrité des données membres. En effet, si l'utilisateur de la classe ne peut pas modifier les données membres directement, il est obligé d'utiliser l'interface (les fonctions membres) pour les modifier, ce qui permet au créateur de la classe d'effectuer des contrôles.

Exercice (QCM)

1. Quand un membre d'une classe est déclaré dans une section *private*, cela signifie :

- O Qu'on ne peut y accéder que depuis l'intérieur de la classe
- O Qu'on ne peut y accéder que depuis l'extérieur de la classe
- O Qu'on peut y accéder depuis l'intérieur et l'extérieur de la classe

2. Dans quel ordre doit-on placer les attributs et les méthodes dans une classe ?

- O D'abord les attributs, après les méthodes
- O D'abord les méthodes, après les attributs
- O L'ordre n'a aucune importance

3. Si on sépare la déclaration de la classe de l'implémentation des méthodes, quel préfixe doit-on mettre devant le nom des méthodes ?

- O NomClasse//
- O NomClasse:
- O NomClasse::

4. Quand on écrit dans la fonction main( ) :

```
Personne Omar;
```

Qui est l'objet, et qui est la classe ?

- O Personne est la classe, Omar est l'objet
- O Omar est la classe, Personne est l'objet
- O Tous les deux sont des objets.

Les objets

1. La création d'objets

En C++, il existe deux façons de créer des objets, c'est-à-dire d'instancier une classe :

- de façon statique
- de façon dynamique

a. La création statique

La création statique d'objet consiste à créer un objet en lui affectant un nom, de la même façon qu'avec une variable :

```
Nom_Classe Nom_Objet;
```

Ainsi, l'objet est accessible grâce à son nom.

b. La création dynamique

La création dynamique d'objets est une création d'objets par le programme lui-même en fonction de ses « besoins ». Les objets ainsi créés ne peuvent pas avoir de nom permettant de les manipuler facilement mais sont repérés par des pointeurs.

La création d'objet dynamique se fait donc par la procédure suivante :

1. définition d'un pointeur vers une classe donnée (celle dont va être instancié l'objet créé dynamiquement) ;
2. création de l'objet « dynamique » grâce au mot clé ***new***, renvoyant l'adresse de l'objet nouvellement créé ;
3. affectation de cette adresse au pointeur.

Voici donc à quoi peut ressembler une création d'objet dynamique en C++ :

```
Nom_Classe * Nom_Pointeur;

Nom_Pointeur = new  Nom_Classe;
```

Grâce à ce pointeur il va désormais être possible de manipuler l'objet « dynamique », c'est-à-dire accéder à ses fonctions et/ou ses données membres.

Remarque

Tout objet créé dynamiquement, c'est-à-dire avec le mot-clé *new* devra impérativement être détruit à la fin de son utilisation grâce au mot clé **delete**. Dans le cas contraire, une partie de la mémoire (celle utilisée par les objets créés dynamiquement) ne sera pas libérée à la fin de l'exécution du programme.

Les objets créés de façon statique n'ont pas besoin d'être détruits, ils sont automatiquement supprimés lorsque le programme ne fonctionne plus dans la portée dans laquelle ils ont été définis.

Le mot clé *delete* s'utilise en le faisant succéder du pointeur vers l'objet à supprimer :

```
delete  Nom_Pointeur;
```

2. Accès aux données membres d'un objet

L'accès aux données membres d'un objet se fait différemment selon que l'objet a été créé de façon statique ou dynamique :

- Pour les objets créés **statiquement**, l'accès aux données membres se fait grâce au nom de l'objet, suivi d'un point (.), puis du nom de la donnée membre :

```
Nom_Objet.Nom_Attribut
```

- Pour les objets créés **dynamiquement**, l'accès aux données membres se fait grâce au nom du pointeur vers l'objet, suivi d'une flèche (->) représentée par un moins (-) et un signe supérieur (>), puis du nom de la donnée membre :

```
Nom_Pointeur -> Nom_Attribut
```

3. Accès aux fonctions membres d'un objet

L'accès aux fonctions membres d'un objet se fait comme pour l'accès aux données membres, c'est-à-dire par un point ou une flèche selon la création de l'objet. La fonction membre est suivie de parenthèses, contenant les paramètres, s'il y en a. L'accès à une fonction membre se fait donc de la façon suivante :

- En statique :

```
Nom_Objet.Nom_Methode(param1, param2,...);
```

- En dynamique :

```
Nom_Objet -> Nom_Methode(param1, param2,...);
```

4. Le pointeur courant this

Le pointeur *this* est une variable système qui permet de désigner l'objet courant. Cette variable est passée en tant que paramètre caché de chaque fonction membre.

Ainsi, lorsque l'on désire accéder à une donnée membre d'un objet à partir d'une fonction membre du même objet, il suffit de faire précéder le nom de la donnée membre par *this*->.

Exemple

```
class Personne
{
        private:
                int _age;
                float _taille;
                char _sexe;
```

```
        public :
                void DefinirAge(int);
};

void Personne::DefinirAge(int _age)
{
        this->_age = _age;
}
```

En réalité, lorsque l'on donne des noms différents aux données membres et aux paramètres dans les fonctions membres, la variable *this* est implicite, cela signifie que l'on n'est pas obligé de mettre *this*-> devant chaque donnée membre.

Dans le cas où une fonction membre doit retourner un pointeur vers l'objet dans lequel elle se trouve, l'utilisation de la variable *this* devient indispensable.

Exemple

```
class Personne
{
        private:
                int _age;
                float _taille;
                char _sexe;
        public :
                void DefinirAge(int);
                Personne * RetournePersonne();
};

void Personne::DefinirAge(int age)
{
        _age = age;
}

Personne * Personne::RetournePersonne()
{
        return this;
}
```

Les constructeurs et les destructeurs

1. La notion de constructeur

Le constructeur est la fonction membre appelée automatiquement lors de la création d'un objet (en statique ou en dynamique). Cette fonction membre est la première fonction membre à être exécutée, il s'agit donc d'une fonction permettant l'initialisation des attributs.

Le constructeur d'un objet porte le même nom que la classe et ne possède aucune valeur de retour (même pas *void*).

La définition de cette fonction membre spéciale n'est pas obligatoire (si les données membres n'ont pas besoin d'être initialisées par exemple) dans la mesure où un *constructeur par défaut* (appelé parfois *constructeur sans arguments*) est défini par le compilateur C++ si la classe n'en possède pas.

Voyons sur un exemple comment se déclare un constructeur :

```
class Personne
{
   private:
       int _age;
       float _taille;
       char _sexe;
   public :
       Personne(int, float, char);    // Constructeur
};

Personne::Personne(int age, float taille, char sexe )
{
   _age = age;
   _taille = taille;
   _sexe = sexe;
}
```

L'appel du constructeur se fait lors de la création de l'objet. De ce fait, l'appel du constructeur est différent selon que l'objet est créé de façon statique ou dynamique :

- **en statique** : le constructeur est appelé grâce à une instruction de déclaration constituée du nom de la classe, suivi par le nom que l'on donne à l'objet et les paramètres entre parenthèses :

```
Personne Ali(12, 1.62, 'M');
```

- **en dynamique** : le constructeur est appelé en définissant un pointeur vers un objet du type désiré puis en lui affectant la valeur retournée par l'opérateur *new* :

```
Personne *ptrAli = new Personne(12, 1.62, 'M');
```

Comme pour n'importe quelle fonction membre, il est possible de surcharger les constructeurs, c'est-à-dire définir plusieurs constructeurs ayant le même nom mais avec des arguments différents en nombre et/ou en type. Ainsi, il sera possible d'initialiser différemment un même objet, selon la méthode de construction utilisée.

Imaginons par exemple que pour l'exemple précédent on veuille pouvoir définir le sexe d'une personne grâce à un entier valant 0 ou 1, ainsi qu'avoir la possibilité de passer en paramètre la lettre 'M' ou 'F', on peut alors définir deux constructeurs pour lesquels le type du troisième argument (le sexe) sera différent. De plus, on va montrer de quelle manière il est possible de contrôler le caractère entré en paramètre :

```
class Personne
{
   private:
      int _age;
      float _taille;
      char _sexe;

   public :
      Personne(int, float, char);   // 1er Constructeur
      Personne(int, float, int);    // 2ème Constructeur
};
```

```
Personne::Personne(int age, float taille, char sexe)
{
   _age = age;
   _taille = taille;
   sexe = toupper(sexe);    // Conversion en majuscule
   if ((sexe=='M')||(sexe=='F'))
        _sexe = sexe;
   else
        cout<<"Erreur d'initialisation";
}

Personne::Personne(int age, float taille, int sexe)
{
   _age = age;
   _taille = taille;
   switch (sexe)
   {
        case 0 :
           _sexe = 'F';
           break;
        case 1:
           _sexe = 'M';
           break;
        default :
           cout<<"Erreur d'initialisation";
           break;
   }
}
```

Enfin, il est possible d'utiliser des *valeurs par défaut* pour les arguments afin d'éviter à avoir à entrer de façon répétitive un ou plusieurs paramètres portant généralement la même valeur.

2. La notion de destructeur

Contrairement aux constructeurs appelés automatiquement lors de la création d'un objet, les destructeurs sont des fonctions membres qui interviennent automatiquement lors de la destruction d'un objet.

Le destructeur est une fonction membre dont la définition ressemble énormément à celle du constructeur, hormis le fait que le nom du destructeur est précédé d'un *tilde* (~) et qu'il ne possède aucun argument.

Les destructeurs ont en général beaucoup moins besoin d'être définis que les constructeurs, c'est donc le destructeur par défaut qui est appelé le cas échéant. Toutefois, lorsque les objets sont chaînés dynamiquement grâce à des pointeurs (lorsqu'une donnée membre d'un objet est un pointeur vers un objet de même type par exemple), ou dans d'autres cas particuliers, la définition d'un destructeur permettant de « nettoyer » l'ensemble des objets peut être indispensable.

Le destructeur, comme dans le cas du constructeur, est appelé différemment selon que l'objet auquel il appartient a été créé de façon statique ou dynamique.

- le destructeur d'un objet créé de façon statique est appelé de façon implicite dès que le programme quitte la portée dans laquelle l'objet est défini.

- le destructeur d'un objet créé de façon dynamique doit être appelé grâce au mot clé *delete*, qui permet de libérer la mémoire occupée par l'objet.

Bien évidemment, un destructeur ne peut être surchargé, ni avoir de valeur par défaut pour ses arguments, puisqu'il n'en a tout simplement pas.

Exemple : Implémentation d'une chaîne de caractères à l'aide de pointeurs

```
#include <iostream>
using namespace std;

class chaine
{
   private:
      char * s; // Pointeur sur la chaîne
   public:
      chaine(void); // Constructeur par défaut.
      chaine(unsigned int); // Deuxième constructeur.
      ~chaine(void); // Destructeur.
};

chaine::chaine(void)
{
   s = NULL; // Initialisation avec le pointeur NULL.
}
```

```
chaine::chaine(unsigned int Taille)
{
   s = new char[Taille+1]; // Allocation de mémoire
   s[0] = '\0'; // Initialisation de la chaîne à "".
}

chaine::~chaine(void)
{
   if (s!=NULL) delete[] s; // Restitution de la
mémoire
}

int main()
{
   chaine s1;
   chaine s2(200); // Une chaîne de 200 caractères
   return 0 ;
}
```

Exercice 1 (QCM)

1. Quel est le rôle du constructeur ?
   - O Nettoyer la mémoire
   - O Déclarer des variables
   - O Initialiser les attributs d'un objet

2. Qu'est-ce qu'un constructeur par défaut ?
   - O Un constructeur qui renvoie `void`
   - O Un constructeur qui n'a aucun paramètre
   - O Un constructeur qui ne contient aucune instruction

3. Une méthode qui commence par un tilde ~ est un ...
   - O Constructeur
   - O Accesseur
   - O Destructeur

4. Peut-on surcharger un constructeur ?
   - O Oui
   - O Non

5. Peut-on surcharger un destructeur ?

   O Oui

   O Non

Exercice 2 : Définir la classe Cercle qui comporte un seul attribut (le rayon) et trois méthodes :

- `Cercle(float)` : constructeur
- `float Perimetre()` qui retourne le périmètre du cercle actuel
- `float Surface()` qui retourne la surface du cercle actuel.

Ecrire le code de la fonction *main( )* qui permet de créer un objet c de type cercle dont le rayon vaut 10 cm puis calcule et affiche son périmètre et sa surface.

Solution

```
#include <iostream>

using namespace std;

const double PI = 3.14;

class  Cercle
{
      private:
            float  Rayon;      // en cm
      public:
            Cercle(float);     // Constructeur
            float  Perimetre();
            float  Surface();
};

Cercle::Cercle(float  r)
{
      Rayon = r;
}

float  Cercle::Perimetre()
{
      return  2*PI*Rayon;
}
float  Cercle::Surface()
```

```
{
      return  PI*Rayon*Rayon;
}

int main( )
{
    Cercle  c(10);
    cout<<"Perimetre = "<<c.Perimetre()<<" cm"<<endl;
    cout<<"Surface = "<<c.Surface()<<" cm2"<<endl;
    return 0;
}
```

Output du programme

```
Perimetre = 62.8 cm
Surface = 314 cm2
```

Les accesseurs et les mutateurs

1. La protection des données membres

L'un des aspects les plus essentiels du concept « orienté objet » est l'encapsulation, qui consiste à définir des étiquettes pour les données et les fonctions membres afin de préciser si celles-ci sont accessibles à partir d'autres classes ou non.

En C++, des données membres portant l'étiquette *private* ne peuvent pas être manipulées directement par les fonctions membres des autres classes. Ainsi, pour pouvoir manipuler ces données membres, le créateur de la classe doit prévoir des fonctions membres spéciales portant l'étiquette *public*.

- Les fonctions membres permettant d'accéder aux données membres sont appelées **accesseurs**, parfois *getter*s (appellation d'origine anglophone).
- Les fonctions membres permettant de modifier les données membres sont appelées **mutateurs**, parfois *setter*s (appellation d'origine anglophone).

2. La notion d'accesseur

Un accesseur est une fonction membre permettant de récupérer le contenu d'une donnée membre protégée. Il doit avoir comme type de retour le type de la variable à renvoyer.

Une convention de nommage veut que l'on fasse commencer de façon préférentielle le nom de l'accesseur par le préfixe *Get*, afin de faire ressortir sa fonction première.

Exemple

```
class Personne
{
        private:
                int _age;
```

```
                float _taille;
                char _sexe;
        public :
                int GetAge();       // Accesseur
};

int Personne::GetAge()
{
        return _age;
}
```

3. La notion de mutateur

Un mutateur est une fonction membre permettant de modifier le contenu d'une donnée membre protégée. Il doit avoir comme paramètre la valeur à assigner à la donnée membre et ne doit pas nécessairement renvoyer de valeur (il possède dans sa plus simple expression le type *void*).

Une convention de nommage veut que l'on fasse commencer de façon préférentielle le nom du mutateur par le préfixe *Set*.

Exemple

```
class Personne
{
        private:
                int _age;
                float _taille;
                char _sexe;
        public :
                void SetAge(int age);    // Mutateur
};

void Personne::SetAge(int age)
{
        _age = age;
}
```

L'intérêt principal d'un tel mécanisme est le contrôle de la validité des données membres qu'il procure. En effet, il est possible (et même conseillé) de tester la valeur que l'on assigne à une donnée membre, c'est-à-dire que l'on effectue un test de validité de la valeur de l'argument avant de l'affecter à la donnée membre.

Exercice (QCM)

1. Qu'est-ce qu'un accesseur ?
   - O Une méthode qui permet de lire un attribut indirectement
   - O Une méthode qui permet de modifier un attribut indirectement
   - O Une méthode qui permet d'accéder à tous les éléments privés de la classe
   - O Une méthode qui affiche tout le contenu de la classe

2. Qu'est-ce qu'un mutateur ?
   - O Une méthode qui permet de lire un attribut indirectement
   - O Une méthode qui permet de modifier un attribut indirectement
   - O Une méthode qui permet d'accéder à tous les éléments privés de la classe
   - O Une méthode qui affiche tout le contenu de la classe

3. Peut-on avoir un accesseur de type void ?
   - O Oui
   - O Non

4. Peut-on avoir un mutateur sans arguments ?
   - O Oui
   - O Non

5. Peut-on surcharger un mutateur ?
   - O Oui
   - O Non

Exercice

Définir la classe Employe qui comporte cinq attributs (matricule, nom, salaire de base, prime, retenue) et trois méthodes :

- `Employe(int, string, float, float, float)` : constructeur de la classe

- **float** `CalculSalaire()` qui renvoie le salaire net de l'employé en utilisant la formule :

  Salaire Net = Salaire de base + prime – retenue

- **void** `Afficher()` qui affiche le matricule, le nom et le salaire net de l'employé.

Solution

```
#include <iostream>
#include <string>

using namespace std;

class Employe
{
   private:
      int _Matricule;
      string _Nom;
      float _SalBase;
      float _Prime;
      float _Retenue;
   public:
      Employe (int, string, float, float, float);
      float CalculSalaire();
      void Afficher();
};
Employe::Employe(int Matricule, string Nom, float
SalBase, float Prime, float Retenue)
{
   _Matricule = Matricule;
   _Nom = Nom;
   _SalBase = SalBase;
   _Prime = Prime;
   _Retenue = Retenue;
}
float Employe::CalculSalaire()
{
   return _SalBase + _Prime - _Retenue;
}
void Employe::Afficher()
{
   cout<<"Matricule = "<<_Matricule<<endl;
   cout<<"Nom   = "<<_Nom<<endl;
   cout<<"Salaire Net = "<<CalculSalaire()<<" TND";
```

```
    cout<<endl;
}
int main( )
{
    Employe E(2525,"Ali Ben Salah",450, 150, 100);
    E.Afficher( );
    return 0;
}
```

Output du programme

```
Matricule = 2525
Nom  = Ali Ben Salah
Salaire Net = 500 TND
```

Exercice

Définir la classe CompteBancaire qui comporte trois attributs (NumCompte, NomClient, Solde) et quatre méthodes :

- `CompteBancaire(int, string, float)` : constructeur de la classe
- `void Deposer(float montant)` qui permet de mettre à jour le solde suite à une opération de versement
- `void Retirer(float montant)` qui permet de mettre à jour le solde suite à une opération de retrait (si le solde actuel le permet)
- `float Consulter()` qui affiche le numéro du compte, le nom du titulaire du compte, et le solde actuel.

Solution

```
#include <iostream>
#include <string>
using namespace std;
class CompteBancaire
{
      private:
            int _NumCompte;
            string _NomClient;
            float _Solde;
      public:
            CompteBancaire (int, string, float);
            void Deposer(float montant);
```

```
            void Retirer(float montant);
            void Consulter();
};
CompteBancaire::CompteBancaire(int NumCompte, string
NomClient, float Solde)
{
      _NumCompte = NumCompte;
      _NomClient = NomClient;
      _Solde = Solde;
}
void CompteBancaire::Deposer(float montant)
{
      _Solde = _Solde + montant;
}
void CompteBancaire::Retirer(float montant)
{
      if (_Solde >= montant)
            _Solde = _Solde - montant;
      else
            cout<<"Solde insuffisant ..."<<endl;
}
void CompteBancaire::Consulter()
{
      cout<<"Numero du compte : "<<_NumCompte<<endl;
      cout<<"Titulaire du compte : "<<_NomClient ;
      cout<<endl;
      cout<<"Solde actuel : "<<_Solde<<" TND"<<endl;
}
int main( )
{
      CompteBancaire C(1024,"Ali Ben Salah",450);
      C.Deposer(200);
      C.Retirer(150);
      C.Consulter( );
      return 0;
}
```

Output du programme

```
Numero du compte : 1024
Titulaire du compte : Ali Ben Salah
Solde actuel : 500 TND
```

Surcharge des opérateurs

Introduction

En C++, les opérateurs ne se différencient des fonctions que syntaxiquement. D'ailleurs, le compilateur traite un appel à un opérateur comme un appel à une fonction. Le langage C++ permet donc de surcharger les opérateurs pour les classes définies par l'utilisateur en utilisant une syntaxe particulière calquée sur la syntaxe utilisée pour définir des fonctions membres normales.

En fait, il est même possible de surcharger les opérateurs du langage (+, -, *, /, ++, --, etc.) pour les classes de l'utilisateur en dehors de la définition de ces classes.

1. Syntaxe générale

La surcharge d'un opérateur se fait en écrivant une fonction qui a la forme suivante :

type operatorOp(opérandes)

où *opérandes* est la liste complète des opérandes.

2. Etude d'un exemple

Dans le programme suivant, on se propose de surcharger les opérateurs d'addition (+) et de multiplication (*) pour couvrir les nombres complexes.

```
#include <iostream>
#include <string>
using namespace std;

class Complexe
{
      private:
            float Reel;
            float Imag;
```

```
	public:
		Complexe(); // constructeur par défaut
		Complexe(float, float); // 2ème constructeur
		Complexe operator+(Complexe);
		Complexe operator*(Complexe);
		void Afficher();
};

Complexe::Complexe()     // création d'un objet vide
{ }

Complexe::Complexe(float re, float im)
{
	Reel = re;
	Imag = im;
}


void Complexe::Afficher()
{
	cout<<Reel<<" + "<<Imag<<"i"<<endl;
}

Complexe Complexe::operator+(Complexe c)
{
	Complexe resultat;
	resultat.Reel = Reel + c.Reel;
	resultat.Imag = Imag + c.Imag;
	return resultat;
}

Complexe Complexe::operator*(Complexe c)
{
	Complexe resultat;
	resultat.Reel = Reel*c.Reel - Imag*c.Imag;
	resultat.Imag = Reel*c.Imag + Imag*c.Reel;
	return resultat;
}

int main ()
{
	Complexe c1(2,2), c2(1,3);
	Complexe c3 = c1 + c2;
	Complexe c4 = c1 * c2;
```

```
        cout<<"c1 = ";
        c1.Afficher();
        cout<<"c2 = ";
        c2.Afficher();
        cout<<"c3 = c1 + c2 = ";
        c3.Afficher();
        cout<<"c4 = c1 * c2 = " ;
        c4.Afficher();
        return 0;
}
```

Output du programme

```
c1 = 2 + 2i
c2 = 1 + 3i
c3 = c1 + c2 = 3 + 5i
c4 = c1 * c2 = -4 + 8i
```

Exercice

1. Ecrire un programme qui définit la classe temps composée de 3 attributs (heures, minutes, secondes) et de 3 méthodes :

- `Temps()` : un premier constructeur
- `Temps(int, int, int)` : un deuxième constructeur
- `void Afficher()` : affiche le temps actuel sous le format universel hh:mm:ss

2. Surcharger l'opérateur « + » pour permettre l'addition de deux temps

3. Surcharger l'opérateur « ++ » afin de pouvoir incrémenter le temps actuel d'une seconde.

Solution

```
#include <iostream>
using namespace std;

class Temps
{
      private:
            int Heures;         // 0..23
            int Minutes;        // 0..59
            int Secondes;       // 0..59
      public:
            Temps (int, int, int);  //constructeur
            void Afficher();
            Temps operator+(Temps);
            Temps operator++();
};

Temps::Temps(int  H, int M, int S)
{
      Heures = H;
      Minutes = M ;
      Secondes = S;
}

void Temps::Afficher()
{
      cout<<Heures<<":"<<Minutes<<":"<<Secondes;
      cout<<endl;
}

Temps Temps::operator+(Temps t)
{
      Heures = Heures + t.Heures;
      Minutes = Minutes + t.Minutes;
      Secondes = Secondes + t.Secondes;
      if (Secondes > 59)
      {
            Minutes++;
            Secondes = Secondes % 60;
      }
      if (Minutes > 59)
      {
            Heures++;
            Minutes = Minutes % 60;
      }
```

```
        if (Heures > 23)
        {
              Heures = Heures % 24;
        }
        return Temps(Heures,Minutes,Secondes);
}
Temps Temps::operator++()
{
        Secondes++;
        if (Secondes > 59)
        {
              Minutes++;
              Secondes = Secondes % 60;
        }
        if (Minutes > 59)
        {
              Heures++;
              Minutes = Minutes % 60;
        }
        if (Heures > 23)
                    Heures = Heures % 24 ;
        return Temps(Heures,Minutes,Secondes);
}
int main( )
{
        Temps t(23,59,59);
        cout<<"t = ";
        t.Afficher();
        t++;
        cout<<"Apres une seconde, t sera = " ;
        t.Afficher();
        cout<<endl;
        Temps u(2,20,22);
        cout<<"u = ";
        u.Afficher();
        Temps v(3,45,33);
        cout<<"v = ";
        v.Afficher();
        Temps w = u + v;
        cout<<"w = u + v = ";
        w.Afficher();
        cout<<endl;
        return 0;
}
```

Output du programme

```
t = 23:59:59
Apres une seconde, t sera = 0:0:0
u = 2:20:22
v = 3:45:33
w = u + v = 6:5:55
```

Fonctions et classes amies

1. Les fonctions amies

Si une fonction F( ) est déclarée comme « amie» (friend) d'une classe C, alors F( ) peut accéder aux champs privés de C.

La déclaration d'une fonction amie se fait de la manière suivante :

```
class C
{
        friend type F(type_param1, type_param2, ...);

        private :
                // définition des données membres de la classe

        public :
                // définition des fonctions membres de la classe
};
```

Exemple

```
#include <iostream>
using namespace std;

class Rectangle
{
      friend Rectangle doubler (Rectangle);

      private:
            float longueur, largeur;    // en cm

      public:
            Rectangle(float L, float l)
            {
                longueur = L;
                largeur = l;
            }
```

```
            void Afficher ()
            {
                cout<<"Longueur = "<<longueur<<" cm";
                cout<<"\tLargeur = "<<largeur<<" cm";
            }
};

Rectangle doubler (Rectangle r)
{
      return Rectangle(r.longueur*2,r.largeur*2) ;
}

int main ()
{
      Rectangle rect1(3,2) ;
      Rectangle rect2 = doubler(rect1);
      cout <<"Rectangle 1: "<<"\t";
      rect1.Afficher();
      cout<<endl;
      cout <<"Rectangle 2: "<<"\t;"
      rect2.Afficher;()
      cout<<endl;
      return 0;
}
```

Output du programme

```
Rectangle 1:    Longueur = 3 cm         Largeur = 2 cm
Rectangle 2:    Longueur = 6 cm         Largeur = 4 cm
```

Dans le programme précédent, la fonction *doubler()*, qui permet de créer un nouveau rectangle dont les dimensions sont les doubles du rectangle passé en tant que paramètre, a pu accéder aux données privées de la classe Rectangle étant donnée qu'elle était déclarée comme fonction amie de cette classe.

2. Les classes amies

Si une classe C2 est déclarée comme « amie» de la classe C1, alors toutes les fonctions membres de C2 peuvent accéder aux champs privés de C1.

La déclaration d'une classe amie se fait de la manière suivante :

```
class C1
{
        friend class C2 ;

        private :
                // définition des données membres de la classe

        public :
                // définition des fonctions membres de la classe
};
```

Exemple

```
#include <iostream>
using namespace std;

class Carre
{
      friend class Rectangle;

      private:
            float Cote ;          // en cm

      public:
            Carre(float x)
            {
                  Cote = x;
            }
};

class Rectangle
{
      private:
            float longueur, largeur ;       // en cm

      public:
            Rectangle ()
            {}

            void Convert(Carre C)
            {
                  longueur = C.Cote;
                  largeur = C.Cote;
            }
```

```
            void Afficher()
            {
                cout<<"Longueur = "<<longueur<<" cm";
                cout<<"\tLargeur = "<<largeur<<" cm";
            }
};

int main()
{
      Carre C1(3.5);
      Rectangle rect1;
      rect1.Convert(C1);
      cout<<"Rectangle 1: "<<"\t ;"
      rect1.Afficher();
      cout<<endl;
      return 0;
}
```

Output du programme

```
Rectangle 1:    Longueur = 3.5 cm       Largeur = 3.5 cm
```

La fonction *Convert( )* permet de transformer un carré de coté x en un rectangle dont la longueur et la largeur sont égales à x.

A noter que dans ce programme la classe Carre considère la classe Rectangle comme une classe amie et lui permet, par conséquent, d'accéder à tous ses membres publics et privés mais la classe Rectangle n'offre pas ce privilège à la classe Carré. De même, la relation d'amitié entre les classes n'est pas transitive.

Exercice

Soit la classe CompteBancaire définie comme suit :

```
class CompteBancaire
{
      private:
            int _NumCompte;
            string _NomClient;
            float _Solde;
```

```
        public:
                CompteBancaire (int, string, float);
                void Deposer(float montant);
                void Retirer(float montant);
                void Consulter();
};
```

Ecrire une fonction externe qui permet de transférer un montant m d'un compte A vers un compte B (si le solde du compte A le permet).

Solution

```
void Transfert(float m, CompteBancaire  A, CompteBancaire &B)
{
        if (A._Solde >= m)
        {
                A.Retirer(m);
                B.Deposer(m);
        }
        else
                cout<<"Solde insuffisant …"<<endl;
}
```

La fonction `Transfert()` doit être déclarée comme amie de la classe CompteBancaire pour pouvoir accéder au solde, sinon, il faut prévoir un accesseur qui permet de retourner le solde d'un compte quelconque.

Héritage entre classes

1. L'héritage simple

L'héritage simple permet de donner à une classe toutes les caractéristiques d'une autre classe. La classe dont elle hérite est appelée *classe mère* ou *classe de base*. La classe elle-même est appelée *classe fille* ou *classe dérivée*.

Les caractéristiques héritées sont les propriétés et les méthodes de la classe de base.

Pour faire un héritage en C++, il faut faire suivre le nom de la classe dérivée par le nom de classe de base dans la déclaration avec les restrictions d'accès aux données.

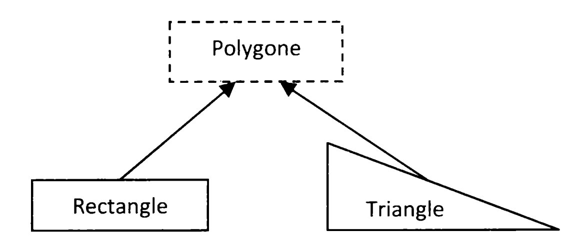
La syntaxe générale est la suivante :

```
class classe_mère
{
        // attributs et méthodes de la classe mère
};

class classe_fille : public|protected|private classe_mère
{
        // attributs et méthodes propres à la classe fille
};
```

Exemple

La classe Polygone englobe toutes les figures géométriques planes fermées formées par la succession d'au moins trois segments appelés côtés comme le triangle et le rectangle. Ces deux formes possèdent des propriétés communes, telles que toutes les deux peuvent être décrites au moyen de seulement deux côtés : largeur et hauteur.



Ceci pourrait être représenté dans le monde des classes avec une classe mère Polygone à partir de laquelle seront dérivées deux classes filles Rectangle et Triangle. La classe Polygone contiendra les membres qui sont communs pour les deux formes à savoir la largeur et la hauteur. Alors que les classes filles contiendront chacune les caractéristiques spécifiques telles que les méthodes de calcul de la surface.

Programme

```
#include <iostream>

using namespace std;

class Polygone
{
	protected:
		float largeur, hauteur;    // en cm

	public:
		Polygone(float l, float h)
		{
			largeur = l;
			hauteur = h;
		}
};

class Rectangle: public Polygone
{
	public:
		Rectangle(float l, float h): Polygone(l,h)
		{}
		float surface()
		{
			return (largeur * hauteur);
		}
};
```

```
class Triangle: public Polygone
{
      public:
            Triangle(float l, float h): Polygone(l,h)
            {}
            float surface ()
            {
                  return (largeur * hauteur)/2 ;
            }
};

int main()
{
   Rectangle R(5,4);
   Triangle T(6,5);
   cout<<"Surface du rectangle = "<<R.surface()<<"
cm2";
   cout<<endl;
   cout<<"Surface du triangle = "<<T.surface()<<"
cm2";
   cout<<endl;
   return 0;
}
```

Output du programme

```
Surface du rectangle = 20 cm2
Surface du triangle = 15 cm2
```

Remarque

La portée *protected* est un droit d'accès qu'on peut classer entre *public* (le plus permissif) et *private* (le plus restrictif). Il n'a de sens que pour les classes qui se font hériter (les classes mères) mais on peut l'utiliser même quand il n'y a pas d'héritage. Sa signification est la suivante : les éléments qui suivent ne seront pas accessibles depuis l'extérieur de la classe, sauf si c'est une classe fille.

Le tableau 14 fournit un récapitulatif des différents modes d'accès aux membres d'une classe selon les fonctions qui veulent accéder à ces membres.

Tableau 14 : Résumé des droits d'accès aux membres d'une classe

	Public	Protected	Private
Fonctions membres de la classe	Oui	Oui	Oui
Fonctions membres des classes dérivées	Oui	Oui	Non
Fonctions externes	Oui	Non	Non

Exercice : Implémenter les 2 classes Personne et Etudiant définies selon la hiérarchie suivante :

Classe Personne
_Nom _Age
Afficher( )

↑

Classe Etudiant
_NCE _Classe
Afficher( )

La fonction Afficher( ) de la classe Personne doit afficher le nom et l'âge de la personne actuelle.

La fonction Afficher( ) de la classe Etudiant doit afficher le numéro de l'étudiant et sa classe en plus de son nom et son âge hérités de la classe mère.

Solution

```
#include <iostream>
#include <string>
using namespace std;

class Personne              // classe mère
{
   protected:
      string _Nom;
      int _Age;

   public:
      Personne(string Nom, int Age)   // constructeur
      {
         _Nom = Nom;
         _Age = Age;
      }
```

```
        void Afficher()
        {
            cout<<"Nom : "<<_Nom<<"\tAge : "<<_Age;
        }
};

class Etudiant : public Personne    // classe fille
{
    private:
        int _NCE;
        string _Classe;

    public:
        Etudiant(string Nom, int Age, int NCE, string
        Classe) : Personne(Nom,Age)
        {
            _NCE = NCE;
            _Classe = Classe;
        }

        void Afficher()
        {
            Personne::Afficher();
            cout<<"\tNum Carte d'etudiant:"<<_NCE;
            cout<<"\tClasse : "<<_Classe<<endl;
        }
};

int main( )
{
    Personne p("Ahmed",20);
    p.Afficher( );
    cout<<endl;
    Etudiant e("Ali",22,1001,"Info 1");
    e.Afficher();
    cout<<endl;
    return 0;
}
```

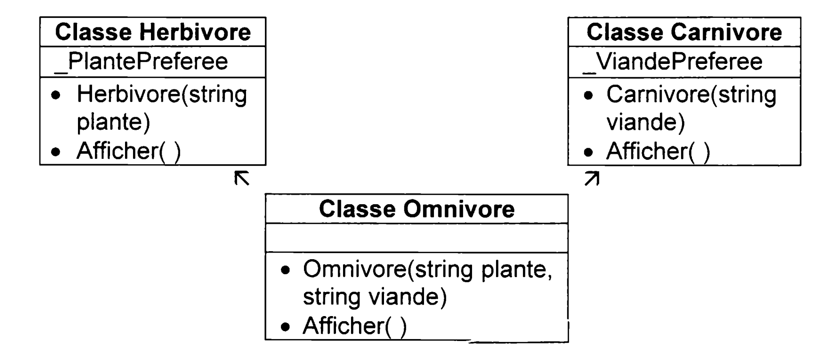
Output du programme

```
Nom : Ahmed   Age : 20
Nom : Ali     Age : 22        Num Carte d'etudiant:1001       Classe : Info 1
```

2. L'héritage multiple

En C++, il est parfaitement possible qu'une classe hérite des membres de plus d'une classe. Ceci se fait simplement en séparant les différentes classes de base par des virgules lors de la déclaration de la classe dérivée.

A titre d'exemple, les animaux qui ont un régime alimentaire omnivore comme l'ours et le corbeau peuvent consommer à la fois de la chair animale et des matières végétales.



Voici l'exemple complet :

```
#include <iostream>
#include <string>

using namespace std;

class Herbivore
{
   protected:
      string _PlantePreferee;

   public:
      Herbivore(string Plante)
      {
            _PlantePreferee = Plante;
      }

      void Afficher()
      {
        cout<<"\tPlante preferee: "<<_PlantePreferee;
```

```
        cout<<endl;
      }
};

class Carnivore
{
   protected:
      string _ViandePreferee ;

   public:
      Carnivore(string Viande)
      {
         _ViandePreferee = Viande;
      }

      void Afficher()
      {
        cout<<"\tViande preferee: "<<_ViandePreferee;
        cout<<endl;
      }
};

class Omnivore : public Herbivore, public Carnivore
{
   public:
      Omnivore(string Plante, string Viande):
      Herbivore(Plante), Carnivore(Viande)
      { }

      void Afficher()
      {
         Herbivore::Afficher( );
         Carnivore::Afficher( );
      }
};

int main( )
{
   Omnivore Ours("Fraise","Poisson");
   cout<<"Les preferences de l'ours"<<endl;
   Ours.Afficher();
   cout<<endl;
   return 0;
}
```

Output du programme

```
Les preferences de l'ours
        Plante preferee: Fraise
        Viande preferee: Poisson
```

3. Le polymorphisme

Un certain nombre de conversions standards sont automatiquement définies entre une classe de base et ses classes dérivées de façon publique. Ainsi, pour une classe A de base et une classe B dérivée de A, des conversions implicites sont définies :

– d'un objet de type B vers un objet de type A ;
– d'un pointeur sur un objet de type B vers un pointeur sur un objet de type A ;
– d'une référence sur un objet de type B vers une référence sur un objet de type A.

Ce type de conversions n'est pas risqué puisqu'un objet de type B est avant tout un objet de type A. Dans le premier cas (celui des objets), c'est une conversion d'*objet* qui est effectuée : l'objet de type B est affecté à l'objet de type A. Dans ce cas, seuls les attributs définis dans A sont pris en compte, les éventuels attributs supplémentaires définis dans B ne sont pas pris en considération. Dans les deux autres cas (ceux des pointeurs et références), c'est seulement une conversion de *type* qui est réalisée : l'objet en lui-même n'est pas affecté. Ce qui implique qu'un pointeur sur le type A peut pointer vers un objet de type B. C'est ce qu'on appelle le *polymorphisme.*

Exemple

```
# include <iostream>

# include <string>

using namespace std;

class Produit              // classe mère
{
   protected:
      string _Nom ;
      float _Prix;
```

```
   public:
      Produit(string Nom, float Prix) // constructeur
      {
          _Nom = Nom ;
          _Prix = Prix;
      }
      void Afficher()
      {
         cout<<"Produit : "<<_Nom ;
         cout<<"\tPrix : "<<_Prix<<" millimes" ;
      }
};

class Boisson : public Produit   //classe fille
{
   private:
      float _Volume;

   public:
      Boisson(string Nom, float Prix, float Vol) :
      Produit(Nom,Prix)
      {
         _Volume = Vol;
      }
};

int main( )
{
      Boisson b("Jus",1300,1);
      Produit * p = &b;
      p->Afficher();
      cout<<endl;
      return 0;
}
```

Output du programme

```
Produit : Jus    Prix : 1300 millimes
```

Exercice (QCM)

1. La classe qui hérite d'une autre classe est aussi appelée la classe...
   - O Mère
   - O Fille
   - O Petite-fille

2. La portée protected empêche l'accès aux méthodes et attributs qui suivent depuis l'extérieur de la classe, sauf...
   - O dans les classes filles
   - O dans la classe mère
   - O dans la fonction main( )

3. Es-ce que le code suivant est légal ?

```
Employe *e1 = NULL;
Enseignant *e2 = new Enseignant();
e1 = e2;
```

   - O Oui
   - O Non
   - O Oui, si Enseignant hérite de Employe

4. Dans un cas d'héritage, dans quel ordre sont exécutés les constructeurs?
   - O Mère puis fille
   - O Fille puis mère
   - O Mère et fille en même temps

5. La classe B hérite de la classe A. Si A possède 3 méthodes et que B en possède 2 qui lui sont propres, combien de méthodes différentes un objet de type B pourra-t-il utiliser ?
   - O 2
   - O 3
   - O 5

Les listes chaînées

1. Structure d'une liste chainée

Une liste chainée contient un ensemble d'éléments de même type qui sont reliés à l'aide de pointeurs. L'adresse de la première cellule est contenue dans un pointeur spécial appelé **tête de liste** (voir figure 10).

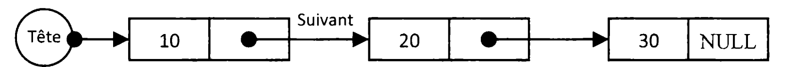


Figure 10 : Structure d'une liste chaînée

La Figure 10 montre comment chaque élément de la liste pointe vers l'élément suivant à part le dernier élément qui pointe vers NULL (une constante prédéfinie qui indique la fin de la liste).

Comparées aux tableaux statiques, les listes chainées sont des structures de données dynamiques qui offrent plusieurs avantages et permettent une gestion plus efficace de la mémoire :

- Il n'est pas nécessaire de définir le nombre d'éléments dès le début, les éléments de la liste peuvent être créés et détruits en cours d'exécution ;
- Les éléments ne sont pas stockés de façon contiguë (côte à côte) comme dans le cas des tableaux ;
- Il est possible d'insérer ou supprimer des éléments sans avoir besoin de décaler les éléments existants.

2. Gestion d'une liste chainée

Chaque élément de la liste est appelé nœud ou cellule. En supposant que la liste contient des entiers, la classe Cellule peut être définie de la façon suivante :

```
// Cellule.h
class Cellule
{
   friend class Liste; //déclaration d'une classe amie
```

```
    private:
        int elem;           // l'information
        Cellule *suiv;      // pointeur vers l'élt suivant
    public:
        Cellule()           // constructeur
        { }
};
```

Par la suite, nous pouvons définir la classe Liste de la façon suivante :

```
// Liste.h
#include <iostream>
using namespace std;

class Liste
{
  private:
        Cellule *tete;      // Tête de la liste
  public:
        Liste()             // constructeur
        {
              tete = NULL;
        }
        bool Vide()
        {
              return (tete == NULL);
        }
        void AjouterDebut(int x);
        void AjouterFin(int x);
        void SupprimerTete();
        void Afficher();
};
void Liste::AjouterDebut(int x)
{
        Cellule *q = new(Cellule);
        q -> elem = x;
        q -> suiv = tete;
        tete = q;
}
void Liste::AjouterFin(int x)
{
        if (Vide())
        {
              Cellule *q = new(Cellule);
```

```
                q -> elem = x;
                q -> suiv = NULL;
                tete = q;
        }
        else
        {
                Cellule *p = tete;
                while (p->suiv != NULL)
                        p = p->suiv;
                Cellule *q = new(Cellule);
                q -> elem = x;
                q ->suiv = NULL;
                p ->suiv = q;
        }
}

void Liste::SupprimerTete()
{
        if (Vide())
        {
                cout<<"La liste est vide !"<<endl;
        }
        else
        {
                Cellule *p = tete;
                tete = tete -> suiv;
                delete p;
        }
}

void Liste::Afficher()
{
        Cellule *p = tete;
        while (p != NULL)
        {
                cout<<p->elem<<"\t";
                p = p->suiv;
        }
}
```

La classe Liste contient une seule donnée membre sous forme de pointeur vers la tête de la liste.

Les objets de type Liste sont gérés à travers les méthodes suivantes :

- Le constructeur `Liste()` qui permet de créer une nouvelle liste vide
- La méthode `Vide()` qui retourne la valeur vraie si la liste ne contient aucun élément
- La méthode `AjouterDebut(int x)` qui permet d'ajouter un élément en tête de liste comme le montre la figure 11 :

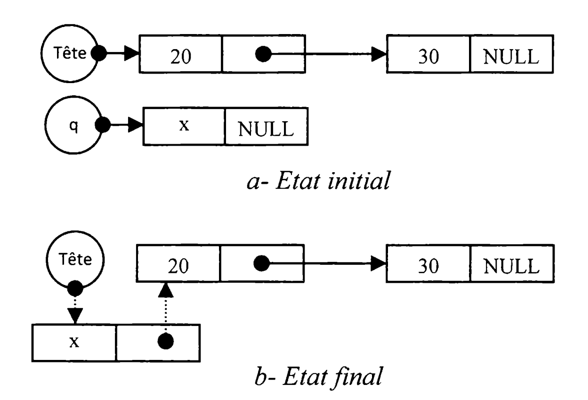


Figure 11 : Ajout d'un élément en tête de liste

- La méthode `AjouterFin(int x)` qui permet d'ajouter un élément en fin de liste comme le montre la figure 12 :

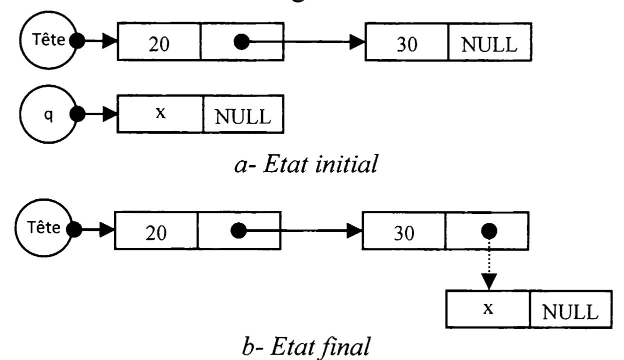
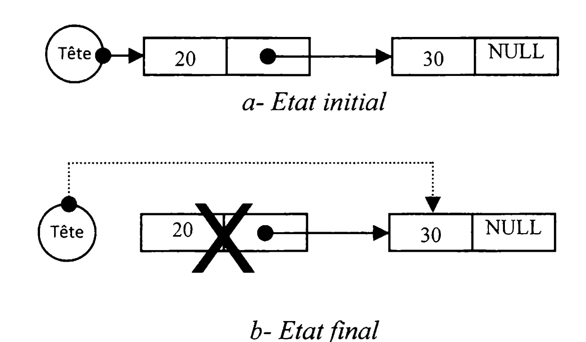


Figure 12 : Ajout d'un élément en fin de liste

- La méthode `SupprimerTete()` qui permet de supprimer le premier élément de la liste comme le montre la figure 13 :



a- Etat initial

b- Etat final

Figure 13 : Suppression de la tête de la liste

- La méthode `Afficher()` qui permet d'afficher les éléments de liste à l'écran.

A titre d'exemple, le programme suivant crée une liste chainée L à laquelle il ajoute deux éléments en début de liste et un troisième à la fin. Ensuite, il supprime la tête la liste et affiche les éléments restants à l'écran :

```
//Liste.cpp
#include "Cellule.h"
#include "Liste.h"
int main()
{
        Liste L;
        L.AjouterDebut(10);
        L.AjouterDebut(20);
        L.AjouterFin(30);
        L.SupprimerTete();
        L.Afficher();
        return 0;
}
```

Output du programme

```
10      30
```

Exercice 1

Ecrire une fonction `int Taille(Liste L)` qui retourne le nombre d'éléments dans la liste L. On suppose que c'est une fonction amie de classe Liste et de la classe Cellule.

Solution

```
int Taille(Liste L)
{
        Cellule *p = L.tete;
        int n = 0;
        while (p != NULL)
        {
                n++;
                p = p->suiv;
        }
        return n;
}
```

Exercice 2

Ecrire une fonction `bool Recherche(Liste L, int x)` qui permet de vérifier si l'élément x existe ou non dans la liste L. On suppose que c'est une fonction amie de classe Liste et de la classe Cellule.

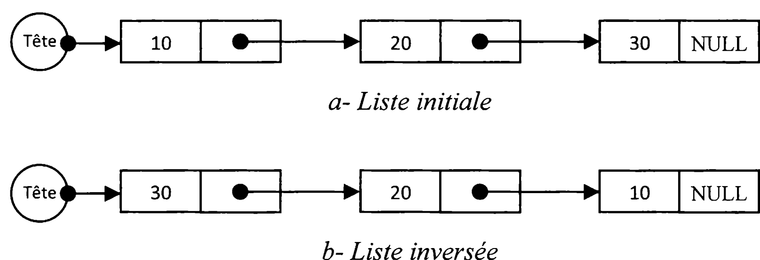
Solution

```
bool Recherche(Liste L, int x)
{
        Cellule *p = L.tete;
        while (p != NULL)
        {
                if (p -> elem == x)
                        return true;
                else
                        p = p -> suiv;
        }
        return false;
}
```

Exercice 3

Ecrire une fonction `void invert(Liste &L)` qui permet d'inverser les éléments de la liste L. On suppose que c'est une fonction amie de classe Liste et de la classe Cellule.

Exemple



a- Liste initiale

b- Liste inversée

Figure 14 : Inversion d'une liste chainée

Solution

```
void Invert(Liste &L)
{
        Cellule *P = L.tete;
        Cellule *Q = NULL;
        Cellule *R;
        while (P != NULL)
        {
                R = new(Cellule);
                R -> elem = P-> elem ;
                R -> suiv = Q;
                Q = R;
                P = P-> suiv;
        }
        L.tete = Q;
}
```

Exercice 4

Ecrire une fonction `void Sort(Liste &L)` qui permet de trier la liste chaînée L dans l'ordre croissant des éléments (utiliser la méthode de tri par bulles). On suppose que c'est une fonction amie de classe Liste et de la classe Cellule.

Solution

```
void Sort(Liste &L)
{
    Cellule *P;
    int NbPerm,temp;
    do
    {
            NbPerm = 0;
            P = L.tete;
            while (P->suiv != NULL)
            {
                  if (P->elem > (P->suiv)->elem)
                  {
                        temp = P->elem;
                        P->elem = (P->suiv)->elem;
                        (P->suiv)->elem = temp;
                        NbPerm++;
                  }
                  P = P->suiv;
            }
    } while (NbPerm > 0);
}
```

3. Les listes chainées circulaires

Une liste chaînée circulaire est une liste particulière dans laquelle le dernier élément pointe sur la tête de la liste (voir figure 15).

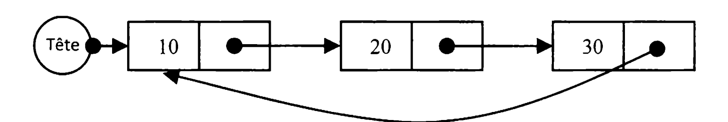


Figure 15 : Structure d'une liste chainée circulaire

Exercice

Ecrire une fonction **void** `Cercler(List &L)` qui permet de transformer la liste chainée L en une liste circulaire. On suppose que c'est une fonction amie de classe Liste et de la classe Cellule.

Solution

```
void Cercler(Liste &L)
{
      Cellule *P = L.tete;
      if (P != NULL)
      {
            while (P-> suiv != NULL)
                  P = P-> suiv;
            P -> suiv = L.tete;
      }
}
```

4. Les listes à chainage double

Ce sont des listes chaînées dont chaque cellule possède deux pointeurs :

- Un pointeur vers l'élément précédent (prec)
- Un pointeur vers l'élément suivant (suiv)

Cette structure permet de parcourir la liste dans les 2 sens et d'accélérer la recherche d'un élément dans la liste (voir figure 16) :

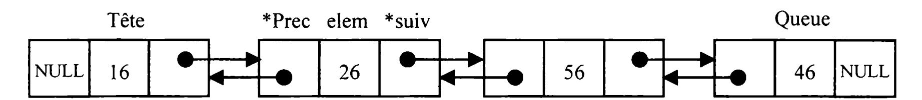


Figure 16 : Structure d'une liste à chainage double

En supposant que la liste contient des entiers, la classe dblCellule peut être définie de la façon suivante :

```
// dblCellule.h
class dblCellule
{
  friend class dblListe; //déclaration d'une classe amie
  private:
    dblCellule *prec;     // pointeur vers l'élt précédent
    int elem;             // l'information
    dblCellule *suiv;     // pointeur vers l'élt suivant
  public:
    dblCellule()          // constructeur
    { }
};
```

Par la suite, nous pouvons définir la classe dblListe de la façon suivante :

```
// dblListe.h
#include <iostream>

using namespace std;

class dblListe
{
  private:
       dblCellule *tete;          // Tête de la liste
       dblCellule *queue;         // Queue de la liste

  public:
       dblListe()                 // Constructeur
       {
              tete = NULL;
              queue = NULL;
       }

       bool Vide()
       {
              return (tete == NULL);
       }

       void AjouterDebut(int x);
       void AjouterFin(int x);
       void ParcoursAvant();
       void ParcoursArriere();
       void Supprimer(int x);
};

void dblListe::AjouterDebut(int x)
{
       dblCellule  *P;
       if (tete == NULL)
       {
              P = new(dblCellule);
              P->prec = NULL;
              P->elem = x;
              P->suiv = NULL;
              tete = P;
              queue = P;
       }
```

```
	else
	{
		P = new(dblCellule);
		P->prec = NULL;
		P->elem = x;
		P->suiv = tete;
		tete->prec = P;
		tete = P;
	}
}

void dblListe::AjouterFin(int x)
{
	dblCellule  *P;
	if (tete == NULL)
	{
		P = new(dblCellule);
		P->prec = NULL;
		P->elem = x;
		P->suiv = NULL;
		tete = P;
		queue = P;
	}
	else
	{
		P = new(dblCellule);
		P->prec = queue;
		P->elem = x;
		P->suiv = NULL;
		queue->suiv = P;
		queue = P;
	}
}

void dblListe::ParcoursAvant()
{
	dblCellule  *P = tete;
	while (P != NULL)
	{
		cout<<P->elem<<"\t";
		P = P->suiv;
	}
}
```

```
void dblListe::ParcoursArriere ()
{
	dblCellule  *P = queue;
	while (P != NULL)
	{
		cout<<P->elem<<"\t";
		P = P->prec;
	}
}
void dblListe::Supprimer(int x)
{
	dblCellule *P;
	if (tete->elem == x)
	{
		P = tete;
		tete = tete ->suiv ;
		tete ->prec = NULL;
		delete P;
	}
	else
	{
		if (queue->elem == x)
		{
			P = queue;
			queue = queue -> prec ;
			queue -> suiv = NULL;
			delete P;
		}
		else
		{
			P = tete;
			while((P != NULL)&&(P->elem != x))
			    P = P -> suiv;
			if (P == NULL)
			    cout<<"Elt introuvable"<<endl;
			else
			{
				(P->prec)->suiv = P->suiv;
				(P->suiv)->prec = P->prec;
				delete P;
			}
		}
	}
}
```

A titre d'exemple, le programme suivant crée une liste L à chainage double à laquelle il ajoute deux éléments en début de liste et un troisième à la fin. Ensuite, il supprime l'élément du milieu et affiche les éléments restants à l'écran :

```
// dblListe.cpp
#include "dblCellule.h"
#include "dblListe.h"
using namespace std;
int main()
{
      dblListe L;
      L.AjouterDebut(26);
      L.AjouterDebut(16);
      L.AjouterFin(36);
      L.Supprimer(26);
      L.ParcoursAvant();
      return 0;
}
```

Output du programme



5. Les piles

Une pile est une suite de cellules allouées dynamiquement (liste chaînée) où l'insertion et la suppression d'un élément se font toujours en tête de liste.

L'image intuitive d'une pile peut être donnée par une pile d'assiettes ou une pile de dossiers à condition de supposer qu'on prend toujours un seul élément à la fois (celui du sommet). On peut résumer les contraintes d'accès par le principe « dernier arrivé, premier sorti » qui se traduit en anglais par : **L**ast **I**n **F**irst **O**ut (voir figure 17).

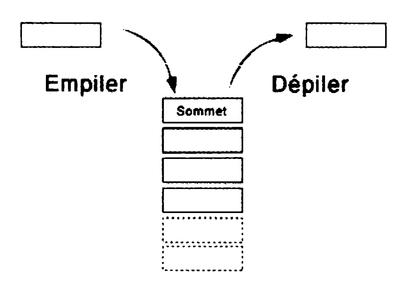


Figure 17 : Structure d'une pile

La structuration d'un objet en pile s'impose lorsqu'on mémorise des informations qui devront être traitées dans l'ordre inverse de leur arrivée. C'est le cas, par exemple, de la gestion des adresses de retour dans l'exécution d'un programme récursif.

En supposant que les éléments de la pile sont des entiers, la classe Pile peut être déclarée de la façon suivante :

```
// Pile.h
#include <iostream>
using namespace std;
class Pile
{
      private:
            Cellule *sommet;   // Sommet de la pile
      public:
            Pile()             // constructeur
            {
                  sommet = NULL;
            }

            bool Vide()
            {
                  return (sommet == NULL);
            }

            void Empiler(int x)
            {
                  Cellule *P = new(Cellule);
                  P->elem = x;
                  P->suiv = sommet;
                  sommet = P;
            }
            void Depiler(void)  // Supprimer le sommet
            {
                  if (Vide())
                    cout<<"La pile est vide "<<endl;
                  else
                  {
                     Cellule  *P = sommet;
                     sommet = sommet -> suiv;
                     delete P;
                  }
            }
```

```
        void Afficher()
        {
                while (!Vide())
                {
                    cout<<sommet->elem<<"\t";
                    Depiler();
                }
        }
};
```

La classe Pile contient une seule donnée membre sous forme de pointeur vers le sommet de la pile.

Les objets de type Pile sont gérés à travers les méthodes suivantes :

- Le constructeur `Pile()` qui permet de créer une nouvelle pile vide
- La méthode `Vide()` qui retourne la valeur vraie si la pile ne contient aucun élément
- La méthode `Empiler(int x)` qui permet d'ajouter l'élément x au sommet de la pile
- La méthode `Depiler()` qui permet de supprimer le sommet de la pile
- La méthode `Afficher()` qui permet d'afficher les éléments de la pile à l'écran.

A titre d'exemple, le programme suivant crée une pile P à laquelle il ajoute les éléments (10, 20, 30, 40, 50). Ensuite, il supprime le sommet de la pile et affiche les éléments restants à l'écran :

```
//Pile.cpp
#include "Cellule.h"
#include "Pile.h"
int main()
{
        Pile P;
        for (int i = 10; i <= 50; i+=10)
                P.Empiler(i);
        P.Depiler();
        P.Afficher();
        return 0;
}
```

Output du programme

```
40          30          20          10
```

6. Les files

Une file est une suite de cellules allouées dynamiquement (liste chaînée) dont les contraintes d'accès sont définies comme suit :

- On ne peut ajouter un élément qu'en dernier rang de la suite ;
- On ne peut supprimer que le premier élément.

L'image intuitive d'une file peut être donnée par la queue à un guichet lorsqu'il n'y a pas de resquilleurs. On peut résumer les contraintes d'accès par le principe « *premier arrivé, premier sorti* » qui se traduit en anglais par : **F**irst **I**n **F**irst **O**ut (voir figure 18).

Exemple

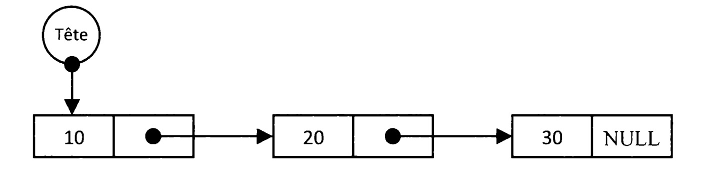


Figure 18 : Structure d'une file

La structuration d'un objet en file s'impose en particulier lorsqu'une ressource doit être partagée par plusieurs utilisateurs : il y a formation d'une *file d'attente*. Un exemple est donné par le spooler d'impression qui est un programme qui reçoit, traite, planifie et distribue les documents à imprimer dans un système multiprogrammé.

En supposant que les éléments de la file sont des entiers, la classe file se déclare de la façon suivante :

```
// File.h
# include <iostream>
using namespace std;
class File
{
      private:
            Cellule *tete;          // tete de la file
```

```
        public:
                File()                        // constructeur
                {
                        tete = NULL;
                }

                bool Vide()
                {
                        return (tete == NULL);
                }

                void Enfiler(int);
                void Defiler();
                void Afficher();
};

void File::Enfiler(int x)    // ajouter x à la fin
{
        if  (Vide())
        {
                Cellule *Q = new(Cellule);
                Q->elem =  x;
                Q->suiv = NULL;
                tete = Q;
        }
        else
        {
                Cellule *R = tete;
                while(R->suiv != NULL)
                        R = R->suiv;
                Cellule *Q = new(Cellule);
                Q -> elem = x;
                Q-> suiv = NULL;
                R-> suiv = Q;
        }
}

void File::Defiler()     // supprimer la tete de la file
{
        if (Vide())
                cout<<"La file est vide !"<<endl;
        else
        {
                Cellule *Q = tete;
```

```
            tete = tete -> suiv;
            delete Q;
        }
}

void File::Afficher()
{
        while (!Vide())
        {
                cout<<tete->elem<<"\t";
                Defiler();
        }
}
```

La classe File contient une seule donnée membre sous forme de pointeur vers la tête de la file.

Les objets de type File sont gérés à travers les méthodes suivantes :

- Le constructeur `File()` qui permet de créer une nouvelle file vide
- La méthode `Vide()` qui retourne la valeur vraie si la file ne contient aucun élément
- La méthode `Enfiler(int x)` qui permet d'ajouter l'élément x à la fin de la file
- La méthode `Defiler()` qui permet de supprimer la tête de la file
- La méthode `Afficher()` qui permet d'afficher les éléments de la file à l'écran.

A titre d'exemple, le programme suivant crée une file F à laquelle il ajoute les éléments (10, 20, 30, 40, 50). Ensuite, il supprime la tête de la file et affiche les éléments restants à l'écran :

```
//File.cpp

#include "Cellule.h"

#include "File.h"

int main()
{
        File F;
```

```
        for (int i = 10; i <= 50; i+=10)
              F.Enfiler(i);
        F.Defiler();
        F.Afficher();
        return 0;
}
```

Output du programme

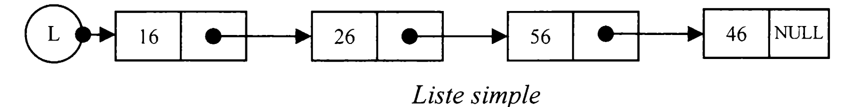


EXERCICES D'APPLICATION

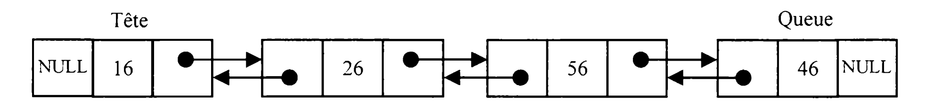
Exercice 1

Ecrire une fonction transform( ) qui transforme une liste simple L en une liste à chainage double.

Exemple



Liste simple

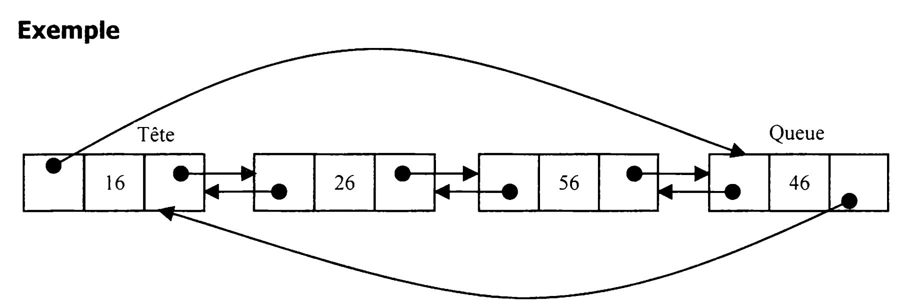


Liste à chainage double

Exercice 2

Ecrire une fonction qui transforme une liste à chainage double L en une liste circulaire.

Exemple



Exercice 3 : Nombres premiers

Ecrire un programme qui permet de construire la liste des nombres premiers inférieurs à un entier n donné.

Pour construire cette liste, on commence par y ajouter tous les nombres entiers de 2 à n dans l'ordre croissant. On utilise ensuite la méthode classique du crible d'Eratosthène qui consiste à parcourir successivement les éléments inférieurs à $\sqrt{n}$ et supprimer tous leurs multiples stricts.

Exercice 4 : Matrices creuses

Les matrices creuses sont des matrices d'entiers comprenant une majorité de 0 comme dans l'exemple suivant :

$$M = \begin{pmatrix} 0 & 2 & 0 & 0 & 0 \\ 1 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 \end{pmatrix}$$

Par souci d'économie, une matrice creuse peut être représentée par une liste chaînée contenant uniquement les éléments non nuls avec leurs indices. Ainsi, la matrice M peut être représentée par la liste suivante :

M = [(2,(0,1)) ; (1,(1,0)) ; (3,(1,2)) ; (6,(3,3)) ; (0,(3,4))]

Les éléments sont rangés par indice croissant (ligne puis colonne). Pour connaître la dimension de la matrice, l'élément d'indice maximal est ajouté en fin de liste, même si celui-ci est nul.

1- Définir la classe Matrice_Creuse
2- Ecrire une fonction qui fait la somme de deux matrices creuses.

Les fichiers

1. Notion de fichier

Dans tous les programmes que nous avons jusqu'à présent développés le stockage des données se fait en mémoire vive qui est volatile et de capacité limitée. Or, la plupart des applications nécessitent une sauvegarde permanente des données.

Pour éviter la perte de ces informations au débranchement de l'ordinateur, on utilise des *fichiers*.

Un fichier est une structure de données formée de cellules contiguës permettant l'implantation d'une suite de données en mémoire secondaire (disque, disquette, CD-ROM, etc.)

Chaque élément de la suite est appelé *article* et correspond généralement à un enregistrement.

Exemples

- liste des employés d'une entreprise
- liste des livres d'une bibliothèque
- état des produits stockés dans un magasin.

2. Gestion des fichiers

En C++, les entrées-sorties sur les fichiers sont réalisées avec des flots. Ce type d'opérations nécessite l'inclusion du fichier d'en-tête `fstream` en plus du fichier d'entrée-sortie standard `iostream`.

Les classes qui permettent de réaliser ces opérations sont :

- La classe **`ofstream`** dédiée aux écritures réalisées dans des fichiers
- La classe **`ifstream`** dédiée aux lectures réalisées dans des fichiers

- La classe **`fstream`** dédiée aux lectures et écritures réalisées dans des fichiers.

Pour créer un fichier de données, il suffit d'instancier un objet de l'une de ces trois classes et spécifier le nom externe du fichier sur disque.

Exemples

- `ofstream fs("C:/f1.dat");   // fichier pour écriture`
- `ifstream fe("C:/f2.dat");   // fichier pour lecture`
- `fstream fes("C:/f3.dat"); // fichier pour lecture et écriture`

Le tableau suivant décrit les principales méthodes offertes par les classes ofstream , ifstream et fstream pour la gestion des fichiers :

Tableau 15 : Principales méthodes de gestion de fichiers

Méthode	Description
open()	Ouverture du fichier pour lecture ou écriture
eof()	Tester si le pointeur a atteint la fin du fichier
write()	Ecriture dans le fichier
read()	Lecture à partir du fichier
close()	Fermeture du fichier

Exemple (création d'un fichier texte)

```
# include <iostream>
# include <fstream>
# include <string>

using namespace std;

int main()
{
   string ligne;
   ofstream f_Ecr ;            //création du fichier
   f_Ecr.open("c:/fich.txt"); //ouverture en écriture
   f_Ecr<<"oeuil pour oeuil"<<endl ; //écriture
   f_Ecr<<"dent pour dent"<<endl ;   //écriture
   f_Ecr.close() ;  //fermeture du fichier
```

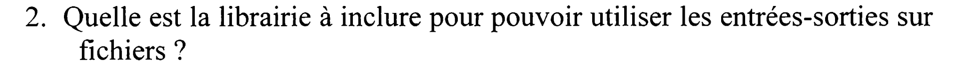
```
    ifstream f_Lec ;
    f_Lec.open("c:/fich.txt"); //ouverture en lecture
    while (!f_Lec.eof())       //parcours du fichier
    {
       getline(f_Lec,ligne) ;  //lecture d'une ligne
       cout<<ligne<<endl;
    }
    f_Lec.close();  //fermeture du fichier
    return 0;
}
```

Output du programme

```
oeuil pour oeuil
dent pour dent
```

Exercice 1 (QCM)

1. Quelle est la librairie à inclure pour pouvoir utiliser les entrées-sorties standards ?
   - O iostream
   - O fstream
   - O string



2. Quelle est la librairie à inclure pour pouvoir utiliser les entrées-sorties sur fichiers ?
   - O iostream
   - O fstream
   - O string

3. Quelle classe faut-il utiliser pour ouvrir un fichier en écriture ?
   - O fstream
   - O ifstream
   - O ofstream

4. Quelle classe faut-il utiliser pour ouvrir un fichier en lecture ?
   - O fstream
   - O ifstream
   - O ofstream

5. Quelle syntaxe faut-il utiliser pour enregistrer le fichier ailleurs que dans le répertoire de l'exécutable (sous Windows) ?
   - O "C:/Documents and Settings/Admin/Bureau/fichier.txt"
   - O "C:\Documents and Settings\Admin\Bureau\fichier.txt"
   - O "C://Documents and Settings//Admin//Bureau//fichier.txt"

6. Comment fermer un flux de lecture/écriture sur fichier ?
   - O flux.open( )
   - O flux.read( )
   - O flux.close( )

Exercice 2

Ecrire un programme qui permet de saisir les données des étudiants (Numéro, Nom, Classe) et les enregistrer dans un fichier. Afficher ensuite la liste des étudiants sous forme d'un tableau.

Solution

```
# include <iostream>
# include <fstream>
# include <string>
using namespace std;

class etudiant
{
   private:
       int Num;
       char Nom[20];
       char Classe[10];

   public:
       etudiant()
       { }
```

```
      etudiant(int n, char nom[], char cl[])
      {
            Num = n;
            strcpy(Nom,nom);
            strcpy(Classe,cl);
      }
      void Afficher()
      {
            cout<<Num<<"\t"<<Nom<<"\t"<<Classe<<endl;
      }
};

int main()
{
   int num;
   char nom[20];
   char classe[10];
   etudiant e;
   // création et ouverture du fichier en écriture
   ofstream ofetud ("C:/fetud.dat", ios::out|ios::binary);
   cout<<"Entrer le numero de l'etudiant (0 pour fin) : ";
   cin>>num;
   while (num != 0)
   {
      cout<<"Entrer le nom : ";
      cin>>nom;
      cout<<"Entrer la classe : ";
      cin>>classe;
      etudiant e(num, nom, classe);
      ofetud.write((char *) &e,sizeof(e)); // écriture
      cout<<"-------------------------------------" ;
      cout<<endl;
      cout<<"Entrer le numero de l'etudiant (0 pour fin):";
      cin>>num;
   }
   ofetud.close();   //fermeture du fichier

   // ouverture du fichier en lecture
   ifstream ifetud ("C:/fetud.dat", ios::in|ios::binary);
   cout<<"NUMERO\tNOM\tCLASSE"<<endl;
   cout<<"------\t---\t------"<<endl;
   ifetud.read((char *) &e, sizeof(e)); // lecture
   while (!ifetud.eof())
   {
      e.Afficher();
```

```
        ifetud.read((char *) &e, sizeof(e));
    }
    ifetud.close();
    cout<<endl;
    return 0;
}
```

Test du programme

```
Entrer le numero de l'etudiant (ou 0 pour fin) : 1001
Entrer le nom : Ali
Entrer la classe : info1
-------------------------------------------------------

Entrer le numero de l'etudiant (ou 0 pour fin): 1002
Entrer le nom : Yahia
Entrer la classe : info1
-------------------------------------------------------

Entrer le numero de l'etudiant (ou 0 pour fin): 1003
Entrer le nom : Karim
Entrer la classe : info2
-------------------------------------------------------

Entrer le numero de l'etudiant (ou 0 pour fin): 0

NUMERO  NOM     CLASSE
------  ---     ------
1001    Ali     info1
1002    Yahia   info1
1003    Karim   info2
```

Notions avancées

1. Les templates

a. A quoi servent les templates ?

Les templates permettent de créer des fonctions et des classes en paramétrant le type de certains de leurs constituants (type des paramètres ou type de retour pour une fonction, type des éléments pour une classe). Les templates permettent d'écrire du code *générique*, c'est-à-dire qui peut servir pour une *famille* de fonctions ou de classes qui ne diffèrent que par la valeur de ces paramètres.

Par exemple, un tableau de *int* ou un tableau de *double* sont très semblables et les fonctions de tri ou de recherche dans ces tableaux sont identiques au type près.

En résumé, l'utilisation des templates permet de « paramétrer » le type des données manipulées.

b. Paramètres des templates

Les paramètres peuvent être de différentes sortes :

- Types simples : *class*, *struct*, types élémentaires comme *int*, *float*, etc.
- Tableaux de taille constante, dont la taille, déduite par le compilateur, peut être utilisée dans l'instanciation du template ;
- Constantes scalaires, c'est-à-dire de type dérivant des entiers (int, long, bool), mais ni double ni float ;
- Templates : La définition d'un template peut être passée à un template, ce qui permet de s'appuyer sur la définition abstraite d'un conteneur par exemple ;
- Pointeurs ou références, à condition que leur valeur soit définie à l'édition de liens ;
- Fonction membre, dont la signature et la classe doivent être aussi passées en paramètres ;

- Membre d'une classe, dont le type et la classe doivent être aussi passés en paramètres du template.

Exemple

La fonction template Max définie ci-dessous peut être appelée avec tout type copiable et comparable avec l'opérateur >.

```
# include <iostream>
# include <string>

using namespace std;

template <typename T> T Max(T a, T b)
{
    return (a > b)? a : b;
}

int main()
{
   int i = Max(3,5); //maximum de 2 entiers
   cout<<"Max(3,5) = "<<i<<endl;
   float x = Max<float>(2,3.2); //maximum de 2 réels
   cout<<"Max(2,3.2) = "<<x<<endl;
   char c = Max('e','b');  //maximum de 2 caractères
   cout<<"Max('e','b') = "<<c<<endl;
   return 0;
}
```

Output du programme

```
Max(3,5) = 5
Max(2,3.2) = 3.2
Max('e','b') = e
```

Dans la ligne `float x = Max<float>(2,3.2)`, nous avons spécifié explicitement le type float pour le type paramétré T car le compilateur ne déduit pas le type de T lorsqu'on passe en même temps un paramètre de type *int* (2) et un paramètre de type *float* (3.2).

2. Les namespaces

Les *namespaces* (espaces de noms) ont été principalement introduits dans la norme définitive de C++ pour gérer les gros projets. Dans ce type de projets, il n'est pas rare d'utiliser plusieurs bibliothèques C++ qui peuvent parfois définir les mêmes identificateurs, ce qui génère des conflits. Avec les *namespaces*, il ne doit plus y avoir de conflits de noms : les déclarations restent cachées dans un *namespace* jusqu'à ce qu'on fasse explicitement appel à lui.

A titre d'exemple, la plupart des déclarations de la bibliothèque standard C++ ont été regroupées dans un *namespace* appelé `std`.

Dans l'exemple qui suit, le nom complet de la fonction `f()` est `MonNameSpace::f()`, selon une syntaxe similaire à celle des méthodes membres de classe. Cependant, afin de ne pas être contraint de désigner la fonction `f()` par rapport à son *namespace* lorsqu'il n'y a pas de risque de conflit, des facilités d'utilisation sont disponibles grâce à la définition d'alias ou l'utilisation du mot-clé `using`.

Exemple

```
namespace MonNameSpace
{
	int f();
	...
} // fin du namespace ;

// définition d'un alias sur le nom du namespace
namespace mon = MonNameSpace;
mon::f(); // Appelle MonNameSpace::f()

// si on souhaite utiliser uniquement la fonction f()
using MonNameSpace::f;
f();// Appelle MonNameSpace:: f()

/* si on veut bénéficier de toutes les déclarations du
namespace */
using namespace MonNameSpace;
```

3. La gestion des exceptions

Une *exception* est l'interruption de l'exécution du programme à la suite d'un événement particulier. Le but des exceptions est de réaliser des traitements spécifiques aux événements qui en sont la cause. Ces traitements peuvent rétablir le programme dans son mode de fonctionnement normal, auquel cas son exécution reprend. Il se peut aussi que le programme se termine, si aucun traitement n'est approprié.

Le langage C++ supporte les exceptions logicielles, dont le but est de gérer les erreurs qui surviennent lors de l'exécution des programmes. Lorsqu'une telle erreur survient, le programme doit *lancer une exception.* L'exécution normale du programme s'arrête dès que l'exception est lancée et le contrôle est passé à un *gestionnaire d'exception.* Lorsqu'un gestionnaire d'exception s'exécute, on dit qu'il a *attrapé l'exception.*

En C++, lorsqu'il faut lancer une exception, on doit créer un objet dont la classe caractérise cette exception et utiliser le mot clé ***throw*** dont la syntaxe est la suivante :

```
throw objet ;
```

où `objet` est l'objet correspondant à l'exception. Cet objet peut être de n'importe quel type, et pourra ainsi caractériser pleinement l'exception. L'exception doit alors être traitée par le gestionnaire d'exception correspondant. On ne peut attraper que les exceptions qui sont apparues dans une zone de code limitée, pas sur tout un programme. On doit donc placer le code susceptible de lancer une exception dans un bloc d'instructions particulier. Ce bloc est introduit avec le mot clé ***try*** :

```
try
{
        // Code susceptible de générer des exceptions...
}
```

Les gestionnaires d'exception doivent suivre le bloc `try`. Ils sont introduits avec le mot clé ***catch*** :

```
catch (classe d'erreur)
{
        // Traitement de l'exception associée à la classe
}
```

Exemple

```
# include <iostream>
using namespace std;

int main()
{
    try
    {
        throw 20;
    }
    catch (int e)
    {
        cout << "Une exception s'est produite"<<endl;
        cout << "Numero de l'exception : "<<e<<endl;
    }
    return 0;
}
```

Output du programme

```
Une exception s'est produite
Numero de l'exception : 20
```

Annexe : Table des caractères ASCII

1. Code ASCII standard

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F		127	7F	□

2. Code ASCII étendu

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
128	80	Ç	160	A0	á	192	C0	└	224	E0	α
129	81	ü	161	A1	í	193	C1	┴	225	E1	ß
130	82	é	162	A2	ó	194	C2	┬	226	E2	Γ
131	83	â	163	A3	ú	195	C3	├	227	E3	π
132	84	ä	164	A4	ñ	196	C4	─	228	E4	Σ
133	85	à	165	A5	Ñ	197	C5	┼	229	E5	σ
134	86	å	166	A6	ª	198	C6	╞	230	E6	µ
135	87	ç	167	A7	º	199	C7	╟	231	E7	τ
136	88	ê	168	A8	¿	200	C8	╚	232	E8	Φ
137	89	ë	169	A9	⌐	201	C9	╔	233	E9	Θ
138	8A	è	170	AA	¬	202	CA	╩	234	EA	Ω
139	8B	ï	171	AB	½	203	CB	╦	235	EB	δ
140	8C	î	172	AC	¼	204	CC	╠	236	EC	∞
141	8D	ì	173	AD	¡	205	CD	═	237	ED	φ
142	8E	Ä	174	AE	«	206	CE	╬	238	EE	ε
143	8F	Å	175	AF	»	207	CF	╧	239	EF	∩
144	90	É	176	B0	░	208	D0	╨	240	F0	≡
145	91	æ	177	B1	▒	209	D1	╤	241	F1	±
146	92	Æ	178	B2	▓	210	D2	╥	242	F2	≥
147	93	ô	179	B3	│	211	D3	╙	243	F3	≤
148	94	ö	180	B4	┤	212	D4	╘	244	F4	⌠
149	95	ò	181	B5	╡	213	D5	╒	245	F5	⌡
150	96	û	182	B6	╢	214	D6	╓	246	F6	÷
151	97	ù	183	B7	╖	215	D7	╫	247	F7	≈
152	98	ÿ	184	B8	╕	216	D8	╪	248	F8	°
153	99	Ö	185	B9	╣	217	D9	┘	249	F9	∙
154	9A	Ü	186	BA	║	218	DA	┌	250	FA	
155	9B	¢	187	BB	╗	219	DB	█	251	FB	√
156	9C	£	188	BC	╝	220	DC	▄	252	FC	ⁿ
157	9D	¥	189	BD	╜	221	DD	▌	253	FD	²
158	9E	₧	190	BE	╛	222	DE	▐	254	FE	■
159	9F	ƒ	191	BF	┐	223	DF	▀	255	FF	□

Bibliographie

Ouvrages

1. ISO/IEC, "C++ International Standard ISO/IEC 14882", 2003.

2. B. Zitouni, "Algorithme & Structure de Données", Centre de Publication Universitaire, Tunis, 2003.

3. Deitel & Deitel, " C++ How to program", Prentice Hall, 2003.

4. Patrick Henry Winston, "On to C++", Addison Wesley, 1995.

5. Juan Soulié, "C++ Langage Tutorial", 2007.

6. Christian Casteyde, "Cours de C/C++", 2003.

7. Henri Garreta, "La langage C++", 2006.

8. Philippe Dosh, "Introduction à la conception objet et à C++", 2001.

9. Marc Mollaret, "C/C++ et programmation objet", Armand Colin Editeur, Paris 1989.

10. Mark Allen Weiss, "Data Structures and Problem Solving using C++", Addison Wesley, 2003.

11. Stanley B. Lippman, "C++ Primer", Addison Wesley, 1995.

Sites web

1. www.cplusplus.com
2. http://www.siteduzero.com
3. www.commentcamarche.net
4. www.wikipedia.org
5. www.msdn.microsoft.org

Table des matières

L'objectif de ce livre est d'expliquer en termes simples et à travers de nombreux exemples et exercices corrigés comment utiliser l'approche orientée objet pour développer des programmes en C++ afin de résoudre différents types de problèmes et manipuler diverses structures de données statiques et dynamiques.

Ouvrage de base pour les étudiants en informatique (Instituts supérieurs des études technologiques, Ecoles d'ingénieurs, Facultés des sciences, Instituts supérieurs de gestion, …).

Ce livre constitue également une référence aux programmeurs débutants et expérimentés qui veulent rafraîchir et compléter leurs connaissances dans le domaine de la programmation orientée objet à travers le langage C++.



Baghdadi ZITOUNI, est professeur agrégé en informatique et technologue à l'Institut supérieur des études technologiques de Ksar-Hellal depuis 1997. Il a également dispensé des cours en informatique dans plusieurs autres institutions universitaires en Tunisie et en Arabie Saoudite.

Outre l'enseignement de l'informatique, et notamment la programmation, il a participé à plusieurs projets de développement de logiciels qui ont abouti à des produits utilisés dans le domaine industriel.

Il est également l'auteur d'un ouvrage sur les algorithmes et les structures de données et de plusieurs articles scientifiques publiés dans des journaux spécialisés.

cv736811c420066su



ISBN : 978-9973-37-644-2 Prix : 12 dinars